2004

DSHOP: Distributed simple hierarchical ordered planner.

Shuyun Lu
University of Windsor

DSHOP:

Distributed Simple Hierarchical Ordered Planner

by

Shuyun Lu

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2004

Your file Votre référence
ISBN: 0-612-92519-6
Our file Notre référence
ISBN: 0-612-92519-6

ABSTRACT

Planning has been an important subject in the area of Artificial Intelligence (AI) for over three decades. Planning is the problem of seeking a series of actions (that is, a plan) that will accomplish a desired goal. Most planning approaches rely on a single processor or a single-agent paradigm. Unfortunately, in a complex world, a single agent may not be sufficient to optimally solve the problem. Distributed Planning is a sub-field of Distributed AI that involves multi-agents working together to solve large planning problems. Distribution may speed up the traditional planning system through parallelism. Hierarchical Task Network (HTN) planning is an AI planning methodology that creates plans by task decomposition. SHOP (Simple Hierarchical Ordered Planner) is a domain-independent HTN planning system designed by Dana Nau *et al.* that plans for tasks in the same order that they will later be executed. This thesis aims at designing and implementing a distributed version of SHOP (that is, DSHOP) and running it on a high performance distributed system called SHARCNET. The implementation is based upon Message Passing Interface (MPI), that is, a library of functions used to achieve parallelism via message-passing. We investigate two approaches to share work between processors: state-copying and state-recomputation. We implemented a state-copying based DSHOP system (DSHOPC), and a state-recomputation based DSHOP system (DSHOPR). We compared these two implementations of DSHOP with the Java version of SHOP on a set of randomly generated artificial domains. A set of experimental results has been used to evaluate the performance of the DSHOP algorithm.

Keywords: Artificial Intelligence (AI), Planning, Hierarchical Task Network (HTN), Simple Hierarchical Ordered Planner (SHOP), distributed planning, message-passing, state-recomputation, state-copying, SHARCNET

To my father, Chunhui Lu

my mother, Zongyue Zhang

my husband, Fei Wang

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to Dr. Scott Goodwin, who has been a constant source of inspiration and encouragement. I also would like to thank Dr. Froduald Kabanza for his support, and guidance that were invaluable in the successful completion of this thesis. I am thankful to Dr. Arunita Jaekel for her insightful comments and constant support. I would like to thank Dr. Myron Hlynka for his comments and suggestions. I also would like to thank Dr. Alioune Ngom and my other committee members for being accommodating when I needed it most. I would like to thank my friends for all the help and support during the completion of this thesis. Also thanks must go to my husband – Fei Wang whose help was a constant source of strength, happiness and encouragement throughout the course of this work. Last, but not least, I would like to thank my parents whose love always keep my hope and confidence.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Artificial Intelligence, or AI for short, "is the art of creating machines that perform functions that require intelligence when performed by people" [29]. People think of AI in different ways, but the essential concept of AI is to create systems that can behave rationally like human beings. AI encompasses a broad range of problems, including diverse topics from machine vision to expert systems.

Unlike lower forms of life, human beings can make plans to achieve their goals. Reasoning and forming plans are also crucial for intelligent machines to deal with real world problems. Planning is an important behavior for any intelligent machine. Planning has been an important subject in AI for over three decades. Planning is the problem of seeking a series of actions that will accomplish a desired goal (that is, a plan) [54]. The planning problem involves many challenges: in the representation of world states, in the specification of actions that modify world states, in techniques for reasoning about the effects of those actions, and in algorithms that search for plans in a search space derived from those actions.

There are some different formulations of the planning problems, such as deterministic planning and non-deterministic planning. In a deterministic planning system, the agent knows explicitly about the effects of every action. STRIPS [20], and SHOP [38] are examples of deterministic planning systems. In a non-deterministic planning, the domain includes actions whose outcomes are uncertain. In many domains, the world can not be completely modeled because of the lack of information. We do not know what is going to happen next. In non-deterministic planning, an action is not a function from one state to

another state, it is a function from one state to a set of states. Examples of non-deterministic planners are UDTPOP [43] and Buridan [30]. Planning under uncertainty is a problem involving AI planning and decision theory. This kind of planning problems can be modeled as Markov decision theoretic planning. In a Markov decision theoretic planning system, the planning domain includes actions that have uncertain effects. The decision maker has incomplete information about the world. There may not be a well-defined goal state [7]. An example of this type of planner is DRIPS [23].

In the rest of this paper, we focus on deterministic planning.

1.1 Basic ideas in planning systems

We use the terms "planner" or "planning system" to refer to software for deterministic planning, and the term "world" to refer to the environment that the planner interacts with.

A planning system needs three essential inputs:

1. the initial state of the world
2. a set of possible actions that can be performed to change the state of the world
3. the goal state of the world

Given these, a planning system can use a suitable planning algorithm to generate a series of actions (i.e., a plan). Then the agent can execute this plan to transfer the world from the initial state to the goal state. So, the output of a planning system is a sequence of actions that can be applied to the initial state to produce a state that satisfies the goal-state description.

The first important issue in planning is how to represent the states, goals, actions and plans. In the traditional context, the initial and goal states are described as sets of predicates, the actions are represented by operators, and a plan is a solution to the planning problem. A plan is represented by a sequence of actions. We may be able to divide the goal into several nearly independent sub-goals, and solve them separately, then combine all the sub-plans together to solve the whole problem (this strategy is not applicable when the combination cost is too high). Using this strategy, a planning system can deal with larger and more complex problems.

An operator consists of two logical formulas: the preconditions, which define the conditions under which the operator may be applied, and the post-conditions, which specify the changes to the state caused by the operator. Predicates that are not mentioned in the post-conditions are assumed not to change during the application of the operator [1]. This kind of representation allows the planner to determine the connections between states and actions, so that the planner can eliminate the irrelevant actions when searching.

The second issue in planning is to choose a suitable planning algorithm. Normally, a planning system needs to accomplish several functions that include choosing the best action based on heuristic search, applying the chosen action, detecting the goal state, detecting dead ends (if after all possibilities have been explored there is no solution, then the planner has reached a dead end), and repairing an almost correct solution [44]. There have been many ways to solve the planning problem, such as logic-based approaches, operator-based approaches (also called the STRIPS approach), temporal approaches (planning with time constraints), case-based approaches (re-use old plans to make new ones), hierarchical planning, distributed planning, etc.

Planners can be domain-dependent or domain-independent. The domain specifies the actions available to the planner. Domain-independent planners are not tied to one particular domain. They can solve problems in different domains. In this paper, we mainly talk about domain-independent planners.

1.2 Representations for planning

A planning system can be described in a formal language, such as STRIPS [20], ADL [41], and PDDL [33].

The classical approach that many planners use today describes states of worlds and operators in the STRIPS language. STRIPS was named after a pioneering planning program known as the STanford Research Institute Problem Solver [45].

In the STRIPS language, each state of the world is represented in terms of a conjunction of positive ground predicates. This means that the description of a state does not necessarily have to be complete. Any ground predicates that are not mentioned in the state can be considered to be false [20].

Goals are also represented in terms of a conjunction of predicates except that goals can also contain variables.

An operator is mainly composed of four parts: preconditions, action description, add list and delete list (together the add list and delete list describe the post-condition):

- Precondition: a conjunction of ground propositions which must be true in the current world before this operator can be applied.
- Action description: a name for the action.

- Add list: a conjunction of new ground propositions which become true after the application of the operator.
- Delete list: a conjunction of old ground propositions which become false after the application of the operator.

Operators specify state transitions, i.e., they change one state into another [19].

Many planning systems describe states and operators in the STRIPS language, such as Blackbox [27], HSP [6], MIPS [15], and STAN [31].

1.3 Planning approaches

There are many different approaches to planning such as case-based planning, graph-based planning, logic-based planning, operator-based planning, hierarchical task-network planning (HTN planning), and many more.

Some of the planning approaches are briefly introduced as follows:

1. Case-based planning: previously generated plans are stored as cases in memory and can be reused to solve similar planning problems in the future. It consists of two steps:
   - Plan matching
   - Plan modification

   CHEF, created by Hammond, is a case-based planner [24]. But researchers found that plan reuse is generally even harder than planning directly, and it would perform better only if two problems are similar enough.
2. Graph-based planning: constructing a compact structure called a Planning Graph before search. The Planning Graph includes all possible actions that

can be performed in each step. Graphplan, developed by Avrim Blum and Merrick Furst [5], was the first planner to use a Planning Graph.

3. Logic-based planning: using logical formulas to specify control formulas that can be used to check against the sequences of states. If the sequence of states violates the control formula then that sequence would be pruned. TLplan, developed by F. Bacchus and F. Kabanza, specifies control knowledge as formulas of temporal logic [2].
4. Operator-based planning: in this kind of planning system, actions are represented as STRIPS-style operators. This approach is also called STRIPS approach. The planning systems that use STRIPS-operators without decompositions are referred to STRIPS-style planners. STRIPS-style planning systems have been developed for more than thirty years.
5. Hierarchical task-network planning (HTN): in this kind of planning system, plans are generated by task decomposition in which the complex tasks would be iteratively decomposed into smaller and smaller subtasks until reaching primitives that can be executed directly. Example planners include UMCP [17], SIPE [55], O-Plan [10], SHOP [38] and more.

In this paper, we mainly talk about HTN planning.

1.4 HTN planning approach

In recent years, some practical planners have adopted an AI planning methodology that generates plans by task decomposition in which the complex tasks are iteratively decomposed into smaller and smaller subtasks until reaching primitives that can be

executed directly. This kind of planning is called hierarchical task network (HTN) planning [19]. Hierarchical decomposition allows us to describe the problem in pieces of a reasonable size. Then we can combine those pieces hierarchically into large plans, without having the trouble of constructing large plans from primitive operators (tasks) (actions that can be directly executed). We will discuss HTN planning approach in more detail in Chapter 2.

There are many sophisticated HTN planners such as NONLIN [51], SIPE and SIPE2 [55], O-Plan [10], UMCP [17], and SHOP [38]. SHOP (i.e., Simple Hierarchical Ordered Planner) is a domain-independent HTN planning system designed by Dana Nau *et al.* [38] that plans for tasks in the same order that they will later be executed. This thesis aims at designing and implementing a distributed version of SHOP.

1.5 Traditional planning assumptions

Planning is difficult because even in the simplest planning problem, it is hard to determine which action should be chosen to change one state to another. Traditional AI planning research has introduced several assumptions and simplifications to make planning feasible [11]:

- "Closed world" assumption: the planning agent is assumed to know everything about the initial state of world with complete certainty. Anything that not explicitly mentioned in the initial state can be presumed false.
- "Instantaneous actions" assumption: actions are executed instantaneously without duration.

- "Deterministic actions" assumption: the planning agent knows explicitly about the effects of every action.
- "Static goals" assumption: the goals are fixed and will not change during the planning process.
- "Static world" assumption: the planning agent is the only source of change in the world.

Classical planning systems use the above assumptions to simplify the problem. Normally, when problems arise due to an assumption that has been made, an extension to the architecture usually exists to work around the problem. For example, in the real world, the environment is normally dynamic, and uncertain. The planning agent cannot make accurate predictions about the effects of every action. One approach to handling this kind of uncertainty is to enumerate the possible states that might arise at execution time and plan for each of them, generating a conditional plan that has alternative courses of action for each state [11].

In fact, researchers have begun to investigate computational models of planning in which one or more of these assumptions is violated. Also, a planner can be integrated with other software modules for solving practical problems.

Although planning systems have not achieved the commercial success like some other areas of artificial intelligence (e.g., neural networks), recently, a number of successful planning applications have been applied to real-world problems. For example, Stephen Smith, Dana Nau, and Thomas Throop use planning technology in the game of contract bridge. John Mark Agosta and David Wilkins's SIPE-2 planner helps evaluate the US Coast Guard's ability to respond to marine oil spills. And SIPE-2 has also been

used in producing military air campaign plans. This planner has been integrated with other software modules to solve the problem.

1.6 Problem and motivation

Most planning approaches rely on a single processor or a single-agent paradigm in which there is a single agent that controls the overall planning process. A few approaches have addressed the problem of distributed planning, using multiple processes or agents to obtain the efficiency of parallel processing [11].

Distributed Planning is a sub-field of distributed AI. Distribution may speed up traditional planning system through parallelism. For large, complex applications, distributed planning systems have many advantages such as system modularity, efficiency, suitability for inherently distributed problems, and reliability.

HTN planning seems suitable to be extended to a distributed environment due to its hierarchical structure. We are proposing a distributed HTN planning system. We expect that it can improve the efficiency by distributed computing.

Our research aims at the objective of developing a distributed version of SHOP (DSHOP). We implemented a state-copying based DSHOP (DSHOPC) and a state-recomputation [8, 42, 43] based DSHOP (DSHOPR). DSHOP ran on SHARCNET [57], and used the message-passing model to allow multiple processes to communicate. In order to take advantage of the fast-growing network technology, we would like to decentralize SHOP into distributed planning-task solvers connected through a network. SHOP and HTN planners in general, have an inherent decomposition structure, in that they decompose a planning task into several subtasks. We expect that this decentralized

version will yield better performance than SHOP, which is already one of the most powerful planning systems in the AI field (In the AIPS'02 international planning competition, SHOP2, a planner derived from SHOP, demonstrated "distinguished performance").

The rest of the proposal is organized as follows: Chapter 2 introduces some related background. Chapter 3 describes the proposed algorithm, and presents an illustrated example of how the algorithm works. Chapter 4 gives the experiments and analyses the results. Chapter 5 discusses some related work in the area of distributed planning. Chapter 6 summarizes the paper and suggests some directions for future work.

CHAPTER 2. BACKGROUND

This chapter introduces some background information related to our work in more detail.

2.1 HTN planning

Hierarchical task network (HTN) planning is an AI planning methodology that plans by task decomposition. In the planning process, the planner decomposes compound tasks into smaller and smaller subtasks until primitive tasks are found that can be executed directly. An HTN planning problem is described as an initial task network that is a set of tasks that need to be accomplished under certain constraints.

2.1.1 Overview of HTN planning

The way we represent the world and actions in HTN planning is similar to STRIPS. Each state of the world is represented by a set of atoms true in that state, and operators (usually called primitive tasks in HTN) have the similar functions as actions. On the other hand, there are still some fundamental differences between them.

HTN planning differs from STRIPS-style planning in two ways. First, their objectives are different. STRIPS-style planners try to find a sequence of actions that can change the initial world state to a goal state, while HTN planners try to accomplish task networks. Second, in STRIPS-style planning systems, a domain consists of a set of operators, while in HTN planning systems, a domain consists of a set of operators and methods. Each method defines a way of how to decompose some task into a set of subtasks, with preconditions that have to be satisfied in order for the method to be applicable [14]. For one task, there may be more than one applicable method, in which case there will be

more than one possible way to decompose the task. Third, STRIPS-style planning proceeds by finding operators whose preconditions are satisfied, while HTN planning plans by task decomposition.

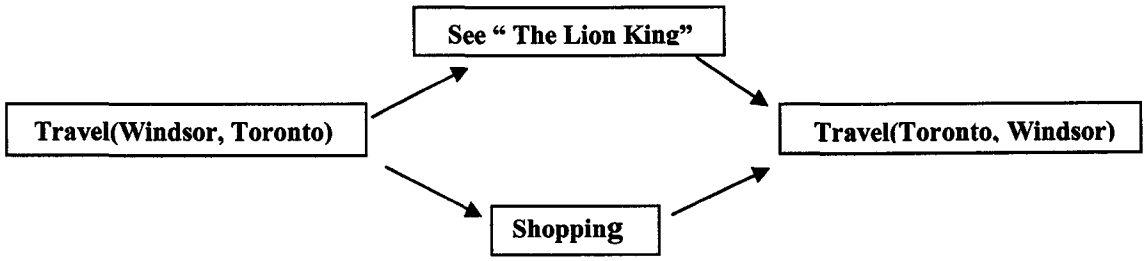

Figure 2.1.1: A simple task network

(this is a modified version of a figure in [18])

In HTN planning system, a task network contains primitive tasks and/or non-primitive tasks. Primitive tasks can be executed directly. Non-primitive tasks cannot be executed directly because they represent activities that involve a set of other tasks. The following example is a modified version of an example in [18]. Figure 2.1.1 represents a task network for a trip from Windsor to Toronto. For instance, the task of traveling to Toronto may have several solutions such as taking the Greyhound bus, take the Via train or driving. The task of taking the 'Greyhound' would involve tasks such as going to the bus station, buying ticket, waiting in the waiting room, and taking the bus; and this solution would work only if several conditions were held, such as availability of tickets, being at the bus station on time, and having enough money for the fare; otherwise, we should consider the other solutions.

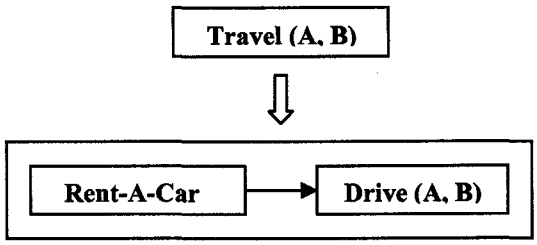

Figure 2.1.2: A method for traveling from city A to city B

(this is a modified version of a figure in [18])

In order to decompose non-primitive tasks into subtasks, the HTN planner defines a set of methods, where each method is a schema for decomposing a task into a set of subtasks. For example, Figure 2.1.2 describes a method for accomplishing Travel(A, B) by achieving tasks Rent-A-Car and Drive(A, B). For each task, there may be more than one applicable method, and thus more than one way to decompose the task into subtasks. The planner may search through these alternative decompositions to find one that is solvable at a lower level.

2.1.2 HTN planning procedure

An HTN planning problem can be represented as a triple $P = <d, I, D>$, where d is the task network (a set of tasks) need to be plan for, I is the initial state (a set of atoms), and D is a set of operators (primitive tasks) and methods (non-primitive tasks) [18]. As we said previously, HTN planning uses task networks instead of goals in STRIPS-style planning. The initial task network is a set of tasks that specifies what we need to accomplish under certain constraints. The planner chooses tasks in the initial task network to decompose into lower-level subtasks until the task network contains only

primitive tasks. Methods tell us how to decompose non-primitive tasks into a set of subtasks. The following is the basic HTN planning procedure [18]:

1. Input a planning problem *P*.
2. If *P* contains only primitive tasks, then resolve the conflicts in *P* and return the result. If the conflicts cannot be resolved, return failure.
3. Choose a non-primitive task *t* in *P*.
4. Choose an expansion for *t*.
5. Replace *t* with the expansion.
6. Use critics to find the interaction among the tasks in *P*, and suggest ways to handle them.
7. Apply one of the ways suggested in step 6.
8. Go to step 2.

"In steps 3 – 5, task decomposition is done by finding a method capable of accomplishing the non-primitive task, and replacing the non-primitive task with the task network produced by the method [18]." For example, the non-primitive task Travel (Windsor, Toronto) in Figure 2.1.1 can be expanded using the method in Figure 2.1.2, then we can get the task network in Figure 2.1.3.

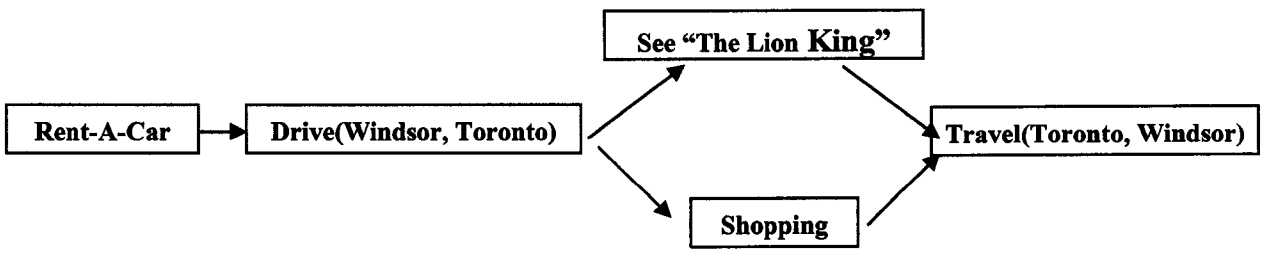

Figure 2.1.3: A decomposition of the task network in Figure 2.1.1

(this is a modified version of a figure in [18, p5])

In step 5, the interactions among tasks may cause conflicts. Critics are used to find and resolve such interactions. In steps 6 and 7, after each expansion, the critics find and resolve interactions. As we can see, critics can detect interactions early to reduce the cost of backtracking [18].

2.1.3 Syntax

HTN planning uses first-order language with some extensions. "The vocabulary of HTN planning language **L** is a tuple <*V, C, P, F, T, N*>, where *V* is an infinite set of variable symbols, *C* is a finite set of constant symbols, *P* is a finite set of predicate symbols, *F* is a finite set of primitive-task symbols(denoting actions), *T* is a finite set of compound-task symbols, and *N* is an infinite set of symbols used for labeling tasks [18]".

- *State*: represented by a set of ground atoms true in that state.
- *Primitive task*: represented by a form do[$f(x_1, \ldots, x_k)$], where $f \in F$ and $x_1, \ldots, x_k$ are terms.
- *Goal task*: represented by a form achieve[*l*], where *l* is a literal.
- *Compound task*: represented by a form perform[$t(x_1, \ldots, x_k)$], where $t \in T$ and $x_1, \ldots, x_k$ are terms.
- *Plan*: a sequence σ of ground primitive tasks.
- *Task network*: represented by a form $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \varnothing]$, where each α_i is a task, $n_i \in N$ is a label for α_i. $\varnothing$ is a Boolean formula including variable binding constraints, ordering constraints, and state constraints.

- *Operator*: represented by a form [operator $f(v_1, \ldots, v_k)$ (pre: $l_1, \ldots, l_m$) (post: $l_1', \ldots, l_n'$)], where f is a primitive task symbol, and $l_1, \ldots, l_m$ are literals describing when f can be executed(i.e., they are preconditions of f), $l_1', \ldots, l_n'$ are literals describing the effects of f, and $v_1, \ldots, v_k$ are the variable symbols used in the literals.
- *Method*: represented by a form (α, d), where α is a non-primitive task, and d is a task network.
- *Planning domain*: represented by a pair <*Op, Me*>, where *Op* is a list of operators, and *Me* is a list of methods.

For each primitive task, there is exactly one operator corresponding to it, while for a non-primitive task, there may be many methods corresponding to it.

2.1.4 Semantics

Now we give the semantics that provide meaning to the syntactic constructs of HTN planning language.

"A semantic structure for HTN planning is a triple $M = <S, F, T>$ [18]":

- $S = 2^{\{\text{all ground atoms}\}}$ is the set of states. Each state in S is a set of atoms true in that state.
- F: $F \times C^* \times S \rightarrow S$ is a partial function for interpreting the actions. Given a primitive task symbol $\in F$, with constant symbols $\in C$, and an input state $\in$ S, F gives the end-up state after the action has been executed.
- T: {ground non-primitive tasks} $\rightarrow 2^{\{groundprimitivetasknetworks\}}$ is a function that maps each non-primitive task α to a set of ground primitive task networks $T(\alpha)$.

No one before had presented a HTN planning algorithm that can be proved to be sound and complete. But based on the above syntax and semantics, K. Erol developed a sound and complete HTN planner: Universal Method-Composition Planner (UMCP) [17].

In general, HTN planning is more expressive than STRIPS-style planning. Every STRIPS-style planning problem can be expressed by an HTN planning problem, but not vice versa. The reasons that HTN planning is more expressive than STRIPS-style planning are [19]:

- Task networks can contain multiple tasks and various constraint formula .
- It can define compound tasks.

Using HTN planning system to solve a planning problem is normally much more complicated than in the previous simple example. One of the complications is that in general, there may be more than one applicable method for a task. In this case, if it is not possible to solve the subtasks produced by one method, it may be necessary to backtrack and try another method [14].

A drawback of HTN planning system is the difficulty of creating a knowledge base for a domain-dependent planning problem.

2.2 SHOP

A plan consists of an ordered set of steps, where each step is an action. Plans can be totally ordered, in which case every step is ordered with respect to every other step, or partially ordered, in which case steps can be unordered with respect to each other. In the past, most AI planning researchers prefer partial-order search to total-order search to

reduce backtracking. Example planners include UCPOP [42], IPP [28], SatPlan, Blackbox [26], SIPE [55], O-Plan [10], UMCP[17], etc. Nevertheless, some researchers have come to realize that total-order forward search has the advantage of making planners more expressive. Example planners include Prodigy [53], TLplan [2], etc.

SHOP, stands for Simple Hierarchical Ordered Planner, is a total-order, domain-independent HTN planning system that plans for tasks in the same order that they will later be executed [38]. To achieve that, SHOP requires the decomposition produced by each method to be a totally ordered set of subtasks.

SHOP is an HTN planning algorithm that creates plans by recursively decomposing tasks (activities that need to be performed) into smaller and smaller subtasks, until primitive tasks are reached (tasks that can be accomplished directly) [38].

SHOP can avoid some of the task-interaction problems encountered in partial-order HTN planning systems. SHOP is much simpler than those partial-order HTN planners since SHOP does not require additional protection conditions to handle partial orderings.

Since SHOP is a total-order forward search planner, "it always knows the complete world-state at each step of the planning process, it can use considerably more expressivities in its domain representations than most AI planners" [36]. For instance, SHOP can use Horn-clause inference, numeric computations, and calls to external programs to evaluate the preconditions of its HTN methods.

SHOP uses first-order language with the notation adapted from Lisp [37]:

- ***Logical symbols***:
    1. Constant symbols, function symbols, and predicate symbols. They are defined like Lisp symbols without question marks at the beginning. For example: car1, move, at-station.
    2. Variable symbols. They are defined like Lisp symbols with question marks at the beginning. For example: ?car
    3. Terms, atoms, ground atoms, conjuncts of atoms, Horn clauses, substitutions, and mgu (most-general unifiers). They are defined similar to Lisp notation.
    4. Task symbols. SHOP has two kinds of task symbols: primitive task symbols defined as Lisp symbols with exclamation points at the beginning, and non-primitive task symbols defined as Lisp symbols without exclamation points at the beginning.
- ***Tasks***: In SHOP, a task is a form $(s\ t_1\ t_2\ \ldots\ t_n)$, which is started with a task symbol(s), and followed by a list of terms $(t_1\ t_2\ \ldots\ t_n)$ as the task's arguments.
- ***Operators***: An operator specifies a way to perform a primitive task. In SHOP, an operator is a form (:operator *h D A c*), where *h* is the head of the operator and should be a primitive task, *D* is the delete list of the operator and consisted of a list of atoms without any variable symbols other than those in *h*, *A* is the add list of the operator and consists of a list of atoms

without any variable symbols other than those in h, and c is a number that means the cost of executing h.

- ***Methods***: Methods are used to decompose non-primitive tasks into more detailed subtasks. In SHOP, a method is the form (:method h C T), where h is the head of the method and should be a compound task, C is the precondition of the method, T is the tail of the method and consists of a task list. Method specifies the way how we can accomplish task h by executing the tasks in T with the same order given in the method.
- ***Axioms***: An axiom is a set of Horn clauses. Axioms are used to find out if the predicates can be inferred from current state.
- ***State***: A state is a set of non-negative ground atoms.
- ***Plans***: In SHOP, a plan consists of a list of heads of ground operator instances (without any variable symbols). Since there are no preconditions for each operator, those operators in the plan are all executable.
- ***Domains***: In SHOP, a domain is consisted of a set of axioms, operators and methods.
- ***Problems***: In SHOP, a planning problem is a triple(S, T, D), where S is a state, T is a task list, and D is a domain.

Given a planning problem (S, T, D)($T = (t_1\ t_2\ \ldots\ t_k)$ is a task list), a plan $P = (p_1\ p_2\ \ldots\ p_n)$ solves (S, T, D) if it satisfies one situation of the following [37]:

1. task list T and plan P are both empty,
2. t_1 is a primitive task, p_1 is a simple plan for t_1 (p_1 should be a operator's head), and (p_2 ... p_n) accomplishes (t_2 ... t_k) from the state S after p_1 applied.
3. t_1 is a compound task, and there is a simple reduction (r_1 ... r_j) of t_1 in S such that P accomplishes (r_1 ... r_j t_2 ... t_k) from S.

If there is a plan that can solve the planning problem (S, T, D), we say this problem is solvable.

The SHOP planning algorithm implements the solution of how to solve a planning problem which we mentioned previously. This algorithm has been proved to be sound and complete.

The following procedure implements the SHOP planning algorithm [38]:

Procedure SHOP(S, T, D) // S is a state, T is a list of tasks, and D is the //knowledge base including methods, operators and // Horn-clause axioms

1. **if** T = nil **then return** nil **endif** // if the task list T is empty, return with nil
2. t = the first task in T // pick the first task in T
3. U = the remaining tasks in T
4. **if** t is primitive and there is a simple plan for t **then**
   // if there is an operator that can accomplish this primitive task
5. non-deterministically choose a simple plan p for t
6. P = SHOP(result(S, p), U, D) // recursively call SHOP after we apply p on S
7. **if** P = FAIL **then return** FAIL **endif**

8. **return** cons(*p*, *P*) // add *p* into plan *P*

9. **else if** *t* is non-primitive and there is a simple reduction of *t* in *S* **then**

10. non-deterministically choose any simple reduction *R* of *t* in *S*

11. **return** SHOP(*S*, append(*R*, *U*), *D*) // replace *t* with its reduction *R*, then //recursively call SHOP

12. **else**

13. **return** FAIL

14. **endif**

end SHOP

SHOP is sound and complete if its precondition-evaluation algorithm is sound and complete. SHOP can be used to resolve complicated real-world planning problems. For instance, the Java version of SHOP has been used as part of HICAP plan-authoring system for Noncombatant Evacuation Operations (NEOs) [35].

2.3 Distributed planning

Complex, dynamic, real-world domains require AI planning researchers to develop systems that are more suitable for realistic planning problems such as weather forecasting system and military operations planning system, in which planning activity is often distributed [11].

Distributed Artificial Intelligence (DAI) has existed as a sub-field of AI for less than two decades. "DAI is concerned with systems that consist of multiple independent entities that interact in a domain" [49].

Distributed Planning is a sub-field of distributed AI. "Distributed planning is the problem of finding a course of action that will help a set of agents in a given initial configuration to collectively satisfy certain desired behavioral constraints." [32].

"Distributed planning is something of an ambiguous term, because it is unclear exactly what is distributed." [12]. We can categorize it into centralized planning for distributed plans, distributed planning for centralized plans, and distributed planning for distributed plans. The emphasis of our research is on distributed planning for centralized plans.

Not all problems are amenable to parallel solution. Problems that are inherently distributed (because of different spatial locations, e.g., a group of companies which have business interaction) or decomposable into sub-applications (e.g., HTN planning problems) are good candidates for distributed planning.

For many kinds of applications, distributed planning systems have significant advantages such as system modularity, efficiency, fast computer architectures, and reliability over large monolithic systems [44].

The distributed planning architecture should provide [44]:

- A mechanism that enables different agents in the system to be coordinated.
- A communication structure that enables information to be passed among agents.
- Distributed versions of planning algorithms.

In the multi agents system, various agents need to be coordinated so that the planning system accomplishes its goal. There are several ways that individual agents can be coordinated to work together effectively including [44]:

- One agent is in charge, this manage agent decomposes the problem into sub-problems, then distributes these sub-problems to other lower-level agents (workers). Agents may communicate with each other to exchange information.
- One agent is in charge and it decomposes the problem into sub-problems, then negotiates with other agents to decide which agent will work on which sub-problem.
- No agent is in charge. There is a single shared goal among all the agents. They cooperate together in generating a plan.
- No agent is in charge, and there is no shared goal among all the agents. They may compete with each other to get the task.

In our implementation of DSHOP, we use the first form to coordinate multiple processors.

When multiple agents work together on a shared planning problem, they can plan with communication with others, or plan without communication [44]. In the first case, agents can communicate with each other during planning procedure. In the second case, the agents work individually on their own tasks without knowing other's processes.

There are two approaches of communication architecture which are:

- Blackboard systems: there is a shared knowledge structure called a blackboard that agents can post and read messages on it.
- Message-passing systems: agents can send messages to others, and receive messages from others.

In the message-passing systems, agents have more information about others than they do in a blackboard system. In our implementation of DSHOP, we use the message-passing approach.

2.4 Message passing

Distributed system involves multiple processes. To share information, avoid conflict, and coordinate, processes must be able to communicate with each other. There are many ways to communicate between remote processes, such as Xerox PARC's ILU (an implementation of CORBA) [25], Java Socket, Knowledge Query Message Language (KQML) [21], Parallel Virtual Machine (PVM) [50], Message-Passing Interface (MPI) [40], etc.

Message-passing is one of the most powerful and widely used paradigms for parallelism on distributed-memory architectures (clusters). The idea of message-passing is not difficult to understand, because people do message passing to exchange information in their daily life. In parallel programming, the reasons for needing message-passing are to exchange data between the parallel tasks, and to synchronize the tasks. If the parallel tasks are completely independent, no message passing is necessary.

The message-passing model assumes a group of processes that have only local memory but are able to communicate with other processes by sending and receiving messages [22]. In basic message-passing, multiple processes coordinate by explicitly sending and receiving messages.

The advantage of message-passing is that the generality of the model of message-passing can be used to program almost any algorithm, and it applies to almost all kinds of computer systems.

2.4.1 Why MPI?

In April 1994, the Message-Passing Interface Forum (MPIF) defined Message Passing Interface (MPI) [56] as a library of functions for message passing among multi-computers and clusters. MPI is one of the first standards for programming parallel processors, and it is the first that is based on message passing [40]. MPI is widely used in parallel programming nowadays.

Normally, MPI can be used in C, FORTRAN, and C++ programs. Our implementation uses the C++-binding MPI.

There are several different types of parallel computing models:

- Data parallel: single instruction, multiple data (SIMD)
- Task parallel: multiple instructions, multiple data (MIMD)
- SPMD: single program, multiple data.

MPI is for MIMD/SPMD parallelism.

The reasons that we choose MPI to develop our parallel programs are:

- MPI provides a powerful, efficient, and portable way to express parallel programs.
- MPI has specified a small group of functions that can be called from C++ programs to achieve parallelism. People do not need to learn a new programming language to write efficient parallel programs. It is not like

KQML, which is a new language for exchanging information. MPI is a standard library.

- MPI is a portable standard for message passing. The message passing model seems to suit this thesis project.
- MPI guarantees the reliability of message transmissions.
- Due to the standardization of MPI, people do not have to care which version of the message passing system is used, as is the case when using the PVM system.
- Our research aims at the objective of developing a distributed version of SHOP, running on SHARCNET [57]. MPI has been installed on SHARCNET.

MPI is small. Although there are many functions in MPI to add flexibility, robustness, efficiency, modularity, or convenience, we can write efficient and effective programs using only six fundamental functions in MPI [22]:

- MPI_Init: initialize MPI
- MPI_Comm_Size: find out how many processes there are
- MPI_Comm_rank: find out which process I am
- MPI_Send: send a message
- MPI_Recv: receive a message
- MPI_Finalize: terminate MPI

MPI is a library of functions. When we write an MPI program in C++, we should include file "mpi.h" like other C++ library files.

In MPI programs, we must call MPI_Init prior to any other MPI calls because this call sets up the MPI environment. At the end of program, we should call MPI_Finalize to close the MPI environment, and after this call, no more MPI calls are allowed.

The message-passing actions are accomplished by MPI_Send and MPI_Recv which are also the most basic and important functions in MPI. MPI_Send sends out a message from one process to another process, and MPI_Recv receives a message from another process.

MPI_Comm_size is used to get the number of processors in current communicator, and MPI_Comm_rank gives the rank number for each processor. If there are n processes executing the program, they will have ranks 0, 1, ..., n-1.

The operating system will give a copy of the executable program on each processor, then every processor can execute its own copy of the program, and different processors' executable program may not be the same since we can take branches by processor ranks. This is the way to write MPI programs.

Sending and receiving data among distributed memory can be expensive. If the amount of information that needs to be sent back and forth is large, it may slow down the performance.

2.5 State-copying and state-recomputation

In a parallel system, when a processor completes its task, it is assigned to another task. The processor has to set up the corresponding state to accomplish the task. There are three ways of setting up the computation state: state-sharing, state-copying, and state-recomputation.

State-sharing requires shared memory. It is not available in a distributed memory environment, such as SHARCNET.

In state-copying, a process sets up its current state by receiving all related data from others. State-copying avoids recomputation. A process can start working on the current state sent by other processes without re-compute from the initial state. Implementation of this model is not difficult because state-copying is independent of operations and only concerned with data structures.

On the one hand, state-copying requires more memory and it may add a lot of communication overhead. Especially when the planning problem is large, the data which need to be exchanged can grow very fast. Normally, the data structure used in a planning system is not a basic data type, and can be quite complicated. Exchanging the user-defined data type in the heterogeneous network is difficult.

Instead of copying a certain state, the processor can re-compute it from the initial state when it is given a certain path. We can use oracles to guide the recomputation. The oracle is the information of non-deterministic choice-points recorded during the original execution [34]. An oracle is just a set of integers. When a process is assigned an oracle, it follows the given oracle, and deterministically takes the corresponding choices. As we said previously, communication in a distributed memory environment can be expensive. On heterogeneous networks, communication bandwidth is low and heterogeneity requires exchanged messages to be in a machine-independent format [34]. The state-recomputation [8, 47, 48] method is motivated by the need for reducing the communication among multiple processors.

For example, in figure 2.5.1, to recompute the state S1 from the Init, a process can simply follow the path (i.e., an oracle) {0, 1} to deterministically choose the corresponding choice at each choice point.

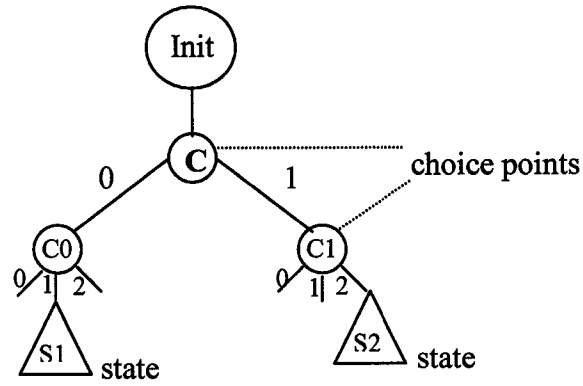

Figure 2.5.1: An example of using oracles to guide the recomputation

The state-recomputation model makes the implementation simpler. By using state-recomputation method, the multiple processors are independent and do not have to share any state, and the processors communicate by exchanging simple data (mainly oracles, which are just sequences of integers) [34]. But since recomputation computes everything from scratch, it also adds a lot of recomputation overhead.

The DELPHI system [8] was the first one to use oracles and recomputation for parallelism. Their research showed that this approach was suited to exploiting Or-Parallelism in Prolog programs. Ehud Shapiro proposed a recomputation-based algorithm and its prototype implementation in Flat Concurrent Prolog [48].

2.6 SHARCNET

The Shared Hierarchical Academic Research Computing Network (SHARCNET) is a network of high-performance computing (HPC) clusters with nodes at five Ontario universities and two colleges [57]. It is a distributed- memory network.

SHARCNET was formally established in June of 2001. It consists of eleven geographically-distributed HPC clusters at academic institutions across Southern Ontario, which include universities of Western Ontario, Guelph, McMaster, Wilfrid Laurier, Windsor, and colleges Fanshawe and Sheridan.

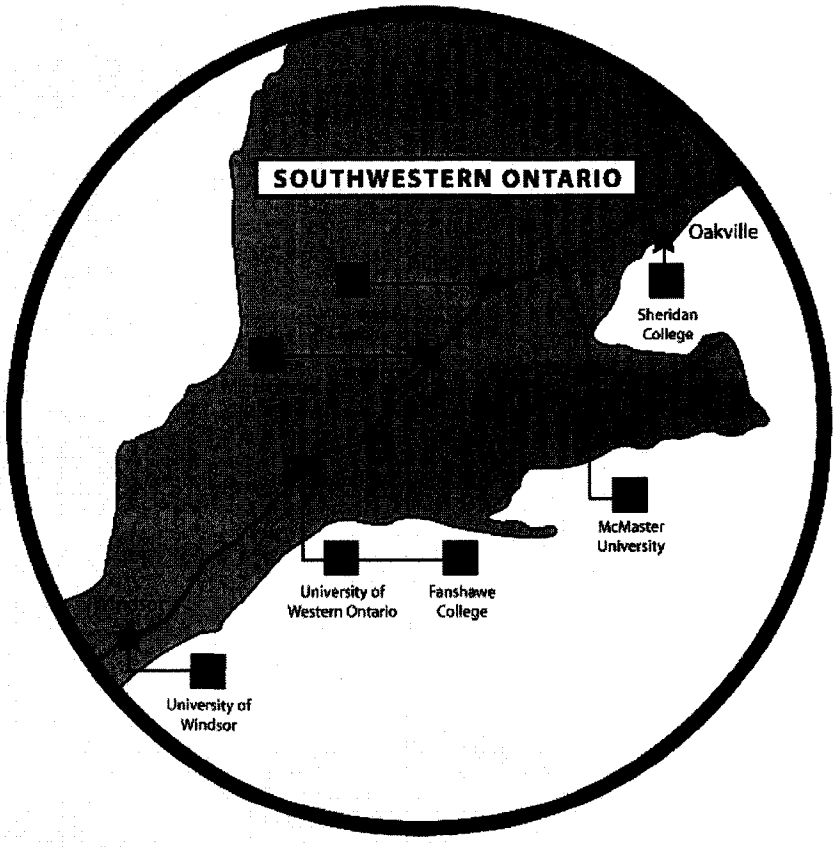

Figure 2.6.1: SHARCNET network [57]

SHARCNET is built on the latest Alpha processors. The clusters consist of four-processor, 833Mhz, Alpha SMP (symmetric multi-processors) systems connected via Quadrics interconnection technology [57]. The Clusters at McMaster University consist of 24 SMPs (96 processors), and the clusters at the University of Guelph consist of 27 SMPs (108 processors). The University of Western Ontario has two clusters, one of 12 SMPs (48 processors) and one of 36 SMPs (144 processors). The University of Windsor and Wilfrid Laurier Universities have smaller clusters (8 processors). We obtained an account at the University of Western Ontario cluster, and did the implementation and experiments on it. The maximum number of processors that we can possibly require for each test run is 144. The more we ask for, the longer we have to wait for all of them to be available.

CHAPTER 3. DESCRIPTION OF DSHOP ALGORITHM

In this chapter, we present a new planning algorithm: the distributed simple hierarchical ordered planning (DSHOP). DSHOP is a distributed version of SHOP. It will be running on SHARCNET [57], using the Message Passing Interface (MPI) [56], that is, a library of functions used to achieve parallelism via message-passing, and setting up the computation state by state-copying (DSHOPC) or state-recomputation (DSHOPR). Like SHOP [38], DSHOP is a domain-independent HTN planning system that plans for tasks in the same order that they will later be executed [38].

3.1 Basic concepts

DSHOP adopts the same syntax and semantics used in SHOP [37]. In addition, we add the following to the DSHOP:

- ***Oracles:*** Basically, oracles are sets of integers that record the non-deterministic choice-points during the original execution. In the recomputation-based DSHOP system, oracles are used to guide the recomputation on different processors.

3.2 DSHOP algorithm

The DSHOP planning system involves multiple processes, and adopts manager/worker architecture. We specify one process as manager that distributes the jobs to the other processes, which are workers. The manager and workers communicate with each other via message-passing using send and receive functions in MPI library. The

manager puts all received messages in a communication buffer using first-in-first-out policy.

The implementation of DSHOP consists of three parts: the manager procedure, the worker procedures, and the DSHOP procedures. The manager procedure only runs on the manager process. It receives the job messages and solution messages from all the workers, and distributes the jobs to the workers. We can also call the manager process as scheduler. There are two procedures running on each worker process: the worker procedure, and the DSHOP procedure. The worker procedure communicates with the manager by MPI. When the worker procedure receives a job message from the manager, it calls the DSHOP procedure to find a solution, then it sends back the solution to the manager. Figure 3.2.1 illustrates the architecture of DSHOP implementation.

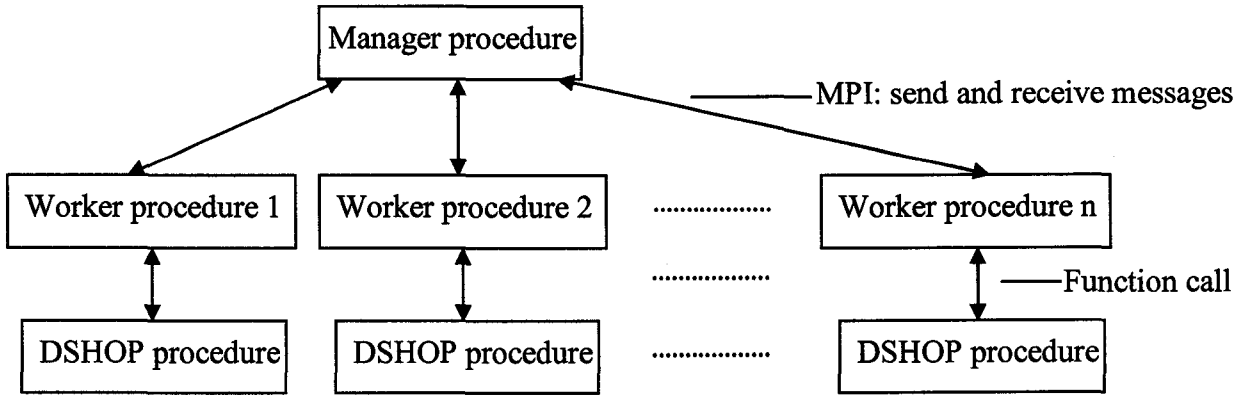

Figure 3.2.1: Architecture of DSHOP implementation

3.2.1 Copying-based DSHOP (DSHOPC)

In the state-copying based DSHOP system (DSHOPC), a worker processor sets up a computation state by copying it from the manager processor.

In the DSHOPC system, if the manager has jobs, it sends a job message to an idle worker. A job message consists of three parts: state, remaining task list, and partial plan. When a worker receives a job message from the manager, it starts to compute using this state set-up. When a worker encounters a new choice-point, it sends a job message back to the manager. This message also consists of three parts: the first part is the current state computed by the worker, the second part is the remaining task list that need to be decomposed, and the third part is the partial plan. When a worker has done its job, it informs the manager.

The following is the description of the DSHOPC algorithm.

The procedure for the manager is shown below:

Procedure Manager() // manager distributes the jobs

1. send an empty message to one of the workers //starting point
2. **do** // loop
3. M = get a message from communication buffer
4. **if** M is a solution **then**
5. send stop message to every worker
6. **return** solution
7. **endif**
8. **if** M is a job message **then**
9. add it to the job list
10. **endif**
11. **while** there is a job J_i and there is an idle worker P_j
12. send J_i to P_j // distribute job to workers

13. delete J_i

14. **endwhile**

15. **if** there is no job and all workers are idle **then** // no solution can be found

16. **return** FAIL

17. **endif**

18. **enddo**

end Manager

The procedure for the workers is shown below:

Procedure Worker()

1. **do**

2. M = received message from manager

3. **if** M is a job message **then**

4. convert job message into three parts: state S, task list T, and partial plan P

5. r = DSHOPC (S, T, P, D) // S is a state, T is a list of tasks, P is partial plan
// D is the domain including operators, methods

6. send r to manager

7. **else if** M is the stop message **then**

8. break

9. **endif**

10. **enddo**

end Worker

The DSHOPC algorithm is shown below:

Procedure DSHOPC (*S*, *T*, *P*, *D*)

1. **if** *T* = nil **then**
2. **return** nil
3. **endif**
4. *t* = the first task in *T*
5. *U* = the remaining tasks in *T*
6. **if** *t* is primitive (i.e., there is an operator for *t*) **then**
7. *N* = number of operators for *t*
8. **if** *N* > 1 **then**
9. convert state, task list and partial plan into MPI data format message *M*

   // state includes all the possible states after the applications of each applicable

   // operator (except for the first one) for *t*

   // plan includes the current partial plan plus all the applicable operators

   // (except for the first one) for *t*
10. send *M* to manager
11. **endif**
12. choose the first operator *op* for *t*
13. add *op* to *P*
14. **return** DSHOPC (*op*(*S*), *U*, *P*, *D*) //after the application of *op*, *S* becomes *op*(*S*)
15. **else if** *t* is non-primitive and there is a simple reduction of *t* **then**
16. *N* = number of reductions of *t*
17. **if** *N* > 1 **then**

18. convert state, task list and partial plan into MPI data format message *M*

19. // task list includes each possible reductions of *t* plus *U*

20. send *M* to manager

21. **endif**

22. choose the first simple reduction *R* of *t*

23. return DSHOPC(*S*, append(*R, U*), *P, D*)

// replace *t* with its reduction *R*, then recursively call DSHOPC

24. **endif**

end DSHOPC

3.2.2 Recomputation-based DSHOP (DSHOPR)

In the state-reomputation based DSHOP system (DSHOPR), a worker processor sets up a computation state by recomputing it from scratch.

In the DSHOPR system, if the manager has jobs, it sends a job (i.e., oracle) to an idle worker. Unlike in DSHOPC, a job message here is a list of integers (i.e., an oracle). When a worker receives an oracle from the manager, it starts to re-compute from the initial state, and determines the choice at each choice-point according to the oracle. When a worker encounters a new choice-point and reaches the end of the oracle, it sends an oracle message back to the manager and follows the first choice to continue the planning process. This oracle message consists of two parts: the first part is the oracle maintained by the worker, and the second part is an integer that indicates the number of extra choices at current choice-point. When a worker has done its job, it informs the manager.

The following is the description of the DSHOPR algorithm.

The procedure for the manager is shown below:

Procedure Manager() // manager distributes the jobs and maintain the oracles

1. send an empty oracle to one of the workers //starting point
2. **do** // loop
3. M = get a message from communication buffer
4. **if** M is a solution **then**
5. send stop message to every worker
6. **return** solution
7. **endif**
8. **if** M is an oracle message **then**
9. generate new oracles
10. **endif**
11. **if** there is an oracle O^i and there is an idle worker P^j **then**
12. send O^i to P^j // distribute job to workers
13. delete O_i
14. **endif**
15. **if** there is no oracle and all workers are idle then // no solution can be found
16. **return** FAIL
17. **endif**
18. **enddo**

end Manager

The procedure for the workers is shown below:

Procedure Worker()

1. **do**
2. O = received message from manager
3. **if** O is an oracle message **then**
4. r = DSHOPR (S, T, D, O) // S is a state, T is a list of tasks, D is the domain // including operators, methods, O is an oracle
5. send r to manager
6. **else if** O is the stop message **then**
7. break
8. **endif**
9. **enddo**

end Worker

The DSHOPR algorithm is shown below:

Procedure DSHOPR (S, T, D, O)

1. **if** T = nil **then**
2. **return** nil
3. **endif**
4. *level* = 0; // current level in the search tree
5. t = the first task in T
6. U = the remaining tasks in T
7. **if** t is primitive (i.e., there is an operator for t) **then**
8. N = number of operators for t

```
9.    if N > 1 then
10.     level++
11.     if level less than the size of O then
12.       choice = get choice from O at level-1
13.     else
14.       choice = 0
15.       add choice to the end of O
16.       send oracle message (O, N) to manager
17.     endif
18.   else
19.     choice = 0
20.   endif
21.   choose the corresponding operator op for t
      // e.g., if choice is 0, choose the first operator; if 2, choose the third one
22.   P = DSHOPR (op(S), U, D,O)//after the application of op, S becomes op(S)
23.   if P = FAIL then return FAIL endif
24.   return cons(p, P) // add p into plan P
25.   else if t is non-primitive and there is a simple reduction of t in S then
26.     N = number of reductions of t
27.     if N > 1 then
28.       level++
29.       if level less than the size of O then
30.         choice = O.GetChoice(level-1)
```

```
31.          else
32.             choice = 0
33.             add choice to the end of O
34.             send oracle message (O, N) to manager
35.          endif
36.       else
37.          choice = 0
38.       endif
39.       choose the corresponding simple reduction R of t in S
40.       return DSHOPR(S, append(R, U), D, O) // replace t with its reduction R,
          // then recursively call DSHOPR
41.    else
42.       return FAIL
43.    endif
end DSHOPR
```

3.2.2 Revised version of DSHOP (DSHOP-*n*)

In DSHOPC, for the problems with deep search trees and with high average branching factors, if the workers send out all the alternative choices to the manager, the number of jobs may be very large, and may cause scheduling problems since there is only one manager processor to deal with all the scheduling. In DSHOPR, for the problems with deep search trees, if the workers send out all the alternative choices to the manager even at the low levels of the trees, the recomputation cost may be very high, and may add too

much communication and scheduling overhead. So we think that maybe we can solve the scheduling problem in DSHOPC and reduce the recomputation cost in DSHOPR by letting the workers only send out the alternatives at the top several levels (*sendoutlevel*) of the search trees and explore the rest of the sub-trees by themselves.

We call this revised version of DSHOP algorithm as DSHOP-*n*, while *n* is the fixed "*sendoutlevel*". From now on, we call the original DSHOPC as DSHOPC-∞, and DSHOPR as DSHOPR-∞ because in the original DSHOP version, we can say the "*sendoutlevel*" is infinite.

In DSHOPC-*n*, the procedures for the manager and worker are the same as in DSHOPC-∞. And in DSHOPR-*n*, the procedures for the manager and worker are the same as in DSHOPR-∞. The different part is in the DSHOP procedure.

As an example, the DSHOPR-*n* procedure is shown below:

Procedure DSHOPR-*n* (*S*, *T*, *D*, *O*) // *S* is a state, *T* is a list of tasks, *D* is the domain // including operators, methods, *O* is an oracle, *n* is // the *sendoutlevel*

1. **if** *T* = nil **then**
2. **return** nil
3. **endif**
4. *level* = 0; // current level in the search tree
5. *t* = the first task in *T*
6. *U* = the remaining tasks in *T*
7. **if** *t* is primitive (i.e., there is an operator for *t*) **then**
8. *N* = number of operators for *t*
9. **if** (*level* < *sendoutlevel* and *N* > 1) **then**

// *sendoutlevel* is fixed here, in our experiments, it is 4

10. *level*++

11. **if** *level* less than the size of *O* **then**

12. choice = get choice from *O* at level-1

13. **else**

14. *choice* = 0

15. add *choice* to the end of *O*

16. send oracle message (*O*, *N*) to manager

17. **endif**

18. choose the corresponding operator *op* for *t*

// e.g., if *choice* is 0, choose the first operator; if 2, choose the third one

19. *P* = DSHOPR-*n*(*op*(*S*), *U*, *D*,*O*)//after the application of *op*, *S* becomes *op*(*S*)

20. **if** *P* = FAIL **then return** FAIL **endif**

21. **return** cons(*p*, *P*) // add *p* into plan *P*

22. **else** // do not send out any jobs to the manager, do backtracking

23. **for** *choice* = *0* to *N*

24. choose the corresponding operator *op* for *t*

25. *P* = DSHOPR-*n*(*op*(*S*), *U*, *D*,*O*)

26. **if** *P* = FAIL **then** continue to try next choice

44. **else return** cons(*p*, *P*) **endif**

45. **endfor**

46. **endif**

47. **else if** *t* is non-primitive and there is a simple reduction of *t* in *S* **then**

48. *N* = number of reductions of *t*

```
49.  if (level < sendoutlevel and N > 1) then
50.    level++
51.    if level less than the size of O then
52.      choice = O.GetChoice(level-1)
53.    else
54.      choice = 0
55.      add choice to the end of O
56.      send oracle message (O, N) to manager
57.    endif
58.   choose the corresponding simple reduction R of t in S
59.    return DSHOPR-n(S, append(R, U), D, O) // replace t with its reduction R,
      // then  recursively call DSHOPR
60.  else // do not send out jobs, do backtracking
61.    for choice = 0 to N
62.       choose the corresponding simple reduction R of t in S
63.       P = DSHOPR-n(S, append(R, U), D, O)
64.       if P = FAIL then continue to try next choice
65.       else return P endif
66.    endfor
67.  endif
68.  else
69.    return FAIL
70.  endif
```

end DSHOPR-*n*

3.3 A simple example of using DSHOP-∞

We use a simple example to illustrate how DSHOPC and DSHOPR systems work.

Suppose we have four operators: o1, o2, o3, and o4:

Operator:	o1	o2	o3	o4
Precondition:	(r1 r3)	(r2 r3)	(r2 r3)	(r3)
Delete list:	(r1)	(r2)	(r3)	(r3)
Add list:	(p1)	(p2)	(p3)	(p4)

Table 3.3.1: Four operators

And we have three methods: init, j1, and j2. Each method may define multiple reductions.

Method:	init	j1	j2	j2
Precondition:		(r1)	(r2)	(p1)
Reduction:	(j1 j2)	(o1 o2)	(o3)	(o4)

Table 3.3.2. Three methods

The initial state is: (r1 r2 r3). The initial task list is (init). Task (init) is the starting point.

Before we explain the DSHOP approach, we first explain how the original SHOP works. Figure 3.3.1 shows the SHOP algorithm working flow.

1. SHOP finds one reductions for (init): (j1 j2). The task list becomes to (j1 j2).
2. For the first task in the task list, (j1), SHOP finds one reduction: (o1 o2). Now the task list becomes to (o1 o2 j2).
3. For the first task in the task list, (o1), SHOP finds one matched operator (o1). After applying the operator (o1) to the state, the state becomes to (r2 r3 p1), and the partial plan is (o1).
4. SHOP finds one matched operator (o2) for task (o2). After applying the operator (o2) to the state, the state becomes to (r3 p1 p2), and the partial plan is (o1 o2).
5. For task (j2), SHOP finds two reductions: (o3) and (o4).
6. SHOP chooses the first reduction (o3). SHOP can not find a simple plan for task (o3) since the precondition (r2 r3) of operator (o3) is not satisfied.
7. SHOP backtracks. It finds one matched operator (o4) for task (o4). After applying the operator (o4) to the current state, the state becomes to (p1 p2 p4), and the partial plan is (o1 o2 o4).
8. The task list now is empty. SHOP returns plan (o1 o2 o4). The procedure stops.

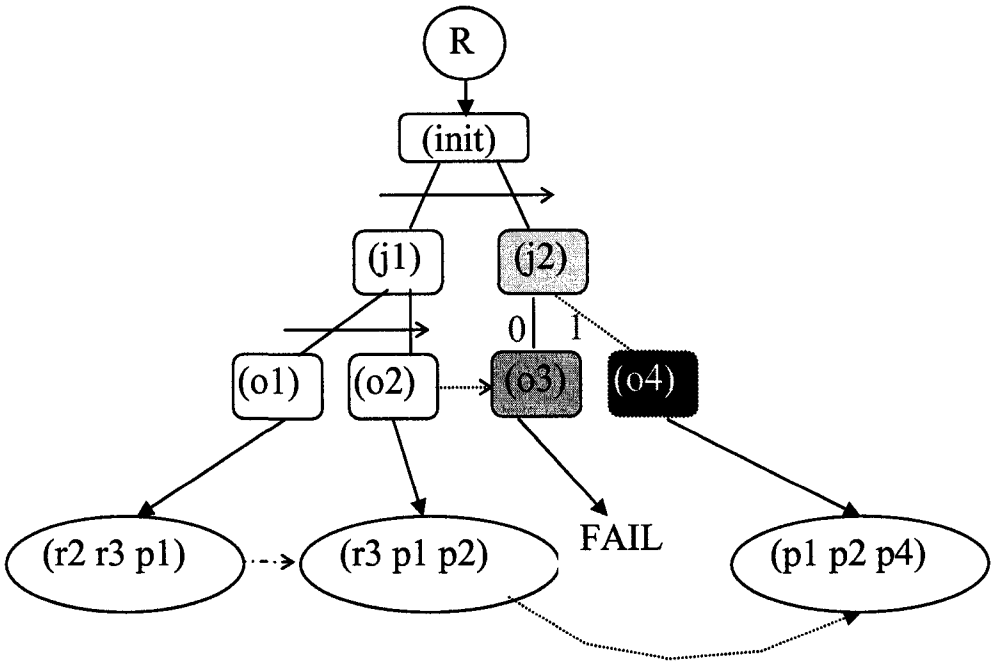

Figure 3.3.1: Example of SHOP working flow

3.3.1 How DSHOPC-∞ works

Instead of backtracking in SHOP, DSHOP-∞ can assign the alternative jobs to other processes. In state-copying based DSHOP-∞, a state is set up by copying the state data from other processes. Figure 3.3.2 shows how the DSHOPC-∞ algorithm works.

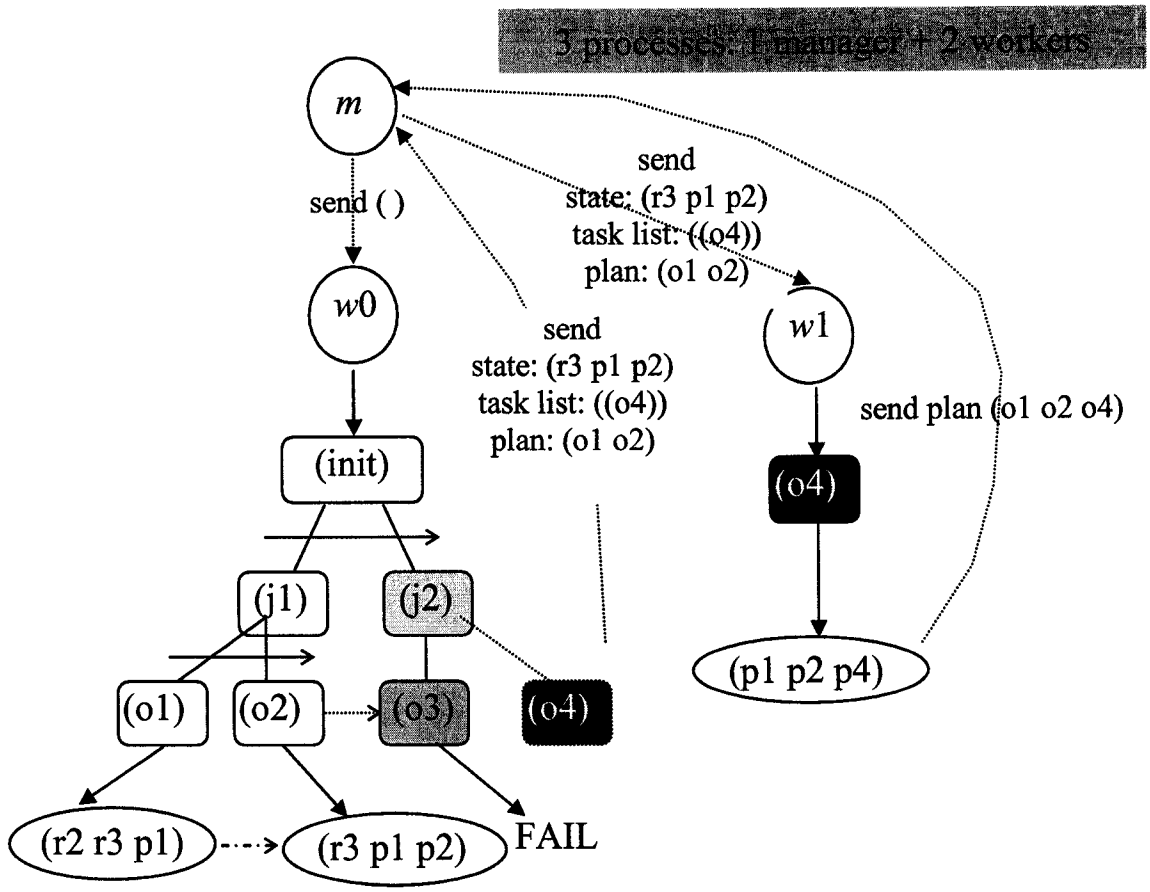

Figure 3.3.2: Example of DSHOPC-∞ working flow

Suppose we have three processors: one manager (*m*) and two workers (*w*0 and *w*1).

1. Firstly, *m* sends an empty message ( ) to *w*0.
2. *w*0 receives an empty message from *m*. It starts working from the initial state. *w*0 finds one reductions for (init): (j1 j2). *w*0's task list becomes (j1 j2).
3. For the first task in the task list, (j1), *w*0 finds one reduction: (o1 o2). Now the task list becomes (o1 o2 j2).

4. For the first task in the task list, (o1), *w*0 finds one matched operator (o1). After applying the operator (o1) to the state, the state becomes: (r2 r3 p1), and the partial plan is (o1).
5. *w*0 finds one matched operator (o2) for task (o2). After applying the operator (o2) to the state, the state becomes: (r3 p1 p2) , and the partial plan is (o1 o2).
6. For task (j2), *w*0 finds two reductions: (o3) and (o4). *w*0 takes the first reduction (o3), and sends a job message (state + task list + partial plan) to *m*.
7. Since state, task list, and plan are not in MPI data format, before sending the message to *m*, *w0* has to convert these data into MPI data format. So *w0* sends the job message (r3 p1 p2 + o4 + o1 o2) to *m*.
8. *m* sends this job message (r3 p1 p2 + o4 + o1 o2) to *w*1.
9. *w*0 can not find a simple plan for task (o3) since the precondition (r2 r3) of operator (o3) is not satisfied. So *w*0 returns fail.
10. Once *w*1 receives the job message (r3 p1 p2 + o4 + o1 o2) from *m*, *w*1 sets up the current state as (r3 p1 p2), task list as (o4), and partial plan as (o1 o2) by converting the message.
11. W1 finds a matched operator (o4) for (o4). After applying the operator (o4) to the current state, the state becomes: (p1 p2 p4). The task list becomes empty. *w*1 returns plan (o1 o2 o4) to *m*.
12. When *m* receives the solution from *w*1, the whole procedure stops.

3.3.2 How DSHOPR-∞ works

Instead of copying a state of computation from other processors, a processor can reach a certain state by recomputing it from the initial state guided by an oracle. The idea of using oracles and recomputation for parallelism was first implemented in the DELPHI system [8]. Now we show how DSHOPR-∞ works.

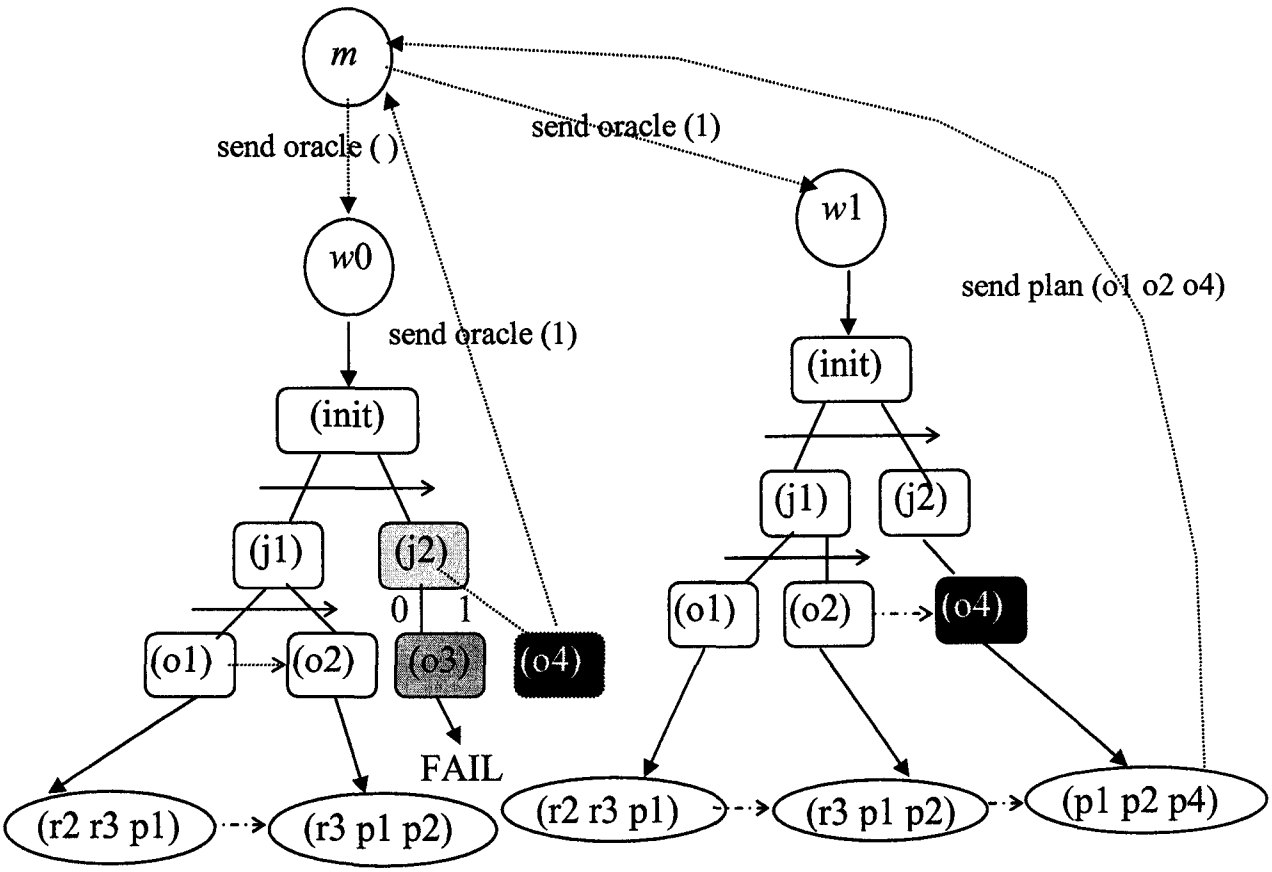

Figure 3.3.3: Example of DSHOPR-∞ working flow

The above figure illustrates how the DSHOP-∞ algorithm works with the recomputation method.

Suppose we have three processors: one manager (m) and two workers ($w0$ and $w1$).

1. Firstly, m sends an empty oracle ( ) to $w0$.
2. $w0$ receives an empty message from m. It starts working from the initial state. $w0$ finds one reductions for (init): (j1 j2). $w0$'s task list becomes (j1 j2).
3. For the first task in the task list, (j1), $w0$ finds one reduction: (o1 o2). Now the task list becomes (o1 o2 j2).
4. For the first task in the task list, (o1), $w0$ finds one matched operator (o1). After applying the operator (o1) to the state, the state becomes: (r2 r3 p1), and the partial plan is (o1).
5. $w0$ finds one matched operator (o2) for task (o2). After applying the operator (o2) to the state, the state becomes: (r3 p1 p2), and the partial plan is (o1 o2).
6. For task (j2), $w0$ finds two reductions: (o3) and (o4). Since the oracle is empty, $w0$ takes the first reduction (o3), and sends an oracle message (1) to m.
7. m sends an oracle message (1) to $w1$.
8. $w0$ cannot find a simple plan for task (o3) since the precondition (r2 r3) of operator (o3) is not satisfied. So $w0$ returns fail.
9. Once $w1$ receives the oracle message (1) from m, $w1$ starts re-computing from the initial state until it reaches the choice-point.
10. Between the two reductions (o3) and (o4), $w0$ chooses the second reduction, which is (o4), according to the oracle. After applying the operator (o4) to the

current state, the state becomes: (p1 p2 p4). The task list becomes empty. *w*1 returns plan (o1 o2 o4) to *m*.

11. When *m* receives the solution from *w*1, the whole procedure stops.

We do not show how DSHOP-*n* works here because there is no big difference between DSHOP-∞ and DSHOP-*n*.

CHAPTER 4. EXPERIMENTS AND EVALUATION

The DSHOP planning system is implemented in C++ linked with MPI library. It is a distributed-memory multiple-instruction multiple-data (MIMD) system. It runs on SHARCNET.

We implemented both copying-based DSHOP-∞ system (DSHOPC-∞) and recomputation-based DSHOP-∞ system (DSHOPR-∞). We compared these two systems to see which one could find a plan faster, and which one had less overhead. We also implemented both copying-based DSHOP-*n* system (DSHOPC-*n*) and recomputation-based DSHOP-*n* system (DSHOPR-*n*). We want to know how the DSHOP performs with variant fixed *"sendoutlevel"*. We also compared these four DSHOP systems with the Java version implementation of SHOP (JSHOP) to see how much speedup could be gained from parallelism.

4.1 Inputs and outputs

In order to do planning in a given planning domain, DSHOP needs to be given knowledge about that domain. Like JSHOP, DSHOP system needs two plain text files as inputs, one of which is the domain definition file that contains operators and methods, one of which is the problem definition file that contains the description of the initial state and a task list that needs to be accomplished. The file format is the same as in JSHOP.

In the domain definition file, a set of operators and methods are defined in the same form as in JSHOP. For instance, the example domain we used in Chapter 3 can be represented as following:

```
;;; Simple artificial domain example for DSHOP

(defdomain artificial-domain01
  (
    ;; operators

    (:operator (!o1)
               ((r1) (r3))
               ((r1))
               ((p1)))

    (:operator (!o2)
               ((r2) (r3))
               ((r2))
               ((p2)))

    (:operator (!o3)
               ((r2) (r3))
               ((r3))
               ((p3)))

    (:operator (!o4)
               ((r3))
               ((r3))
               ((p4)))

    ;; methods

    (:method (init)
             ()
             ((j1) (j2)))

    (:method (j1)
             ((r1))
             ((!o1) (!o2)))

    (:method (j2)
             ((r2))
             ((!o3)))

    (:method (j2)
             ((p1))
             ((!o4)))))
```

In the problem definition file, the initial state and a task list are given as below:

```
;; name: artificial-domain01-problem01.shp
;;

(defproblem ad-test1
            ((r1) (r2) (r3))  //the initial state
            ((init))          //the task list
 )
```

Given the above domain and problem files as input, DSHOP generates the following plan as output:

```
(!o1)
(!o2)
(!o4)
```

4.2 Domains

Like SHOP, DSHOP is a domain-independent planning system. It can be applied on different domains. We proposed to do experiments on three domains: BlocksWorld [39], Logistics [52], and artificial domains.

At first, we tried to use BlocksWorld [39] and Logistics [52] planning domains to analyze the performance of DSHOP because these two domains had been used in the experiments of SHOP, but we found that DSHOP could not get better performance on these two domains. The reason that DSHOP could not gain much speedup on BlocksWorld problem and Logistics problem was simply because there was no backtracking during the planning process.

We ran tests for the BlocksWorld domain on a set of 100 randomly generated problems. Each problem consists of $N = 5, 10, 15, \ldots, 100$ blocks to be relocated. And we also performed tests for the Logistics transportation domain on a set of 100 randomly generated problems. Each problem consists of $N = 10, 15, \ldots, 60$ packages to be delivered [37]. From those experiments, we observed that due to the very good search-control knowledge defined in the domain descriptions of the BlocksWorld and Logistics problems, the planning process did not involve any backtracking. It meant that in BlocksWorld and Logistics problems, at each step of the planning process, SHOP could determinatively choose a simple plan for a primitive task or a simple reduction for a

compound task. Since at each step of the planning process, there was no alternative needing to be explored, even if we allocated more than one processor for each run of DSHOP, only one processor could get the job, and the others all remained idle. That is why DSHOP could not do better than SHOP on BlocksWorld and Logistics problems with the well-defined domain descriptions.

So, does that mean that DSHOP cannot improve on SHOP at all?

To study whether the use of DSHOP algorithm leads to better performance, we carried out various experiments based on a set of randomly generated artificial domains that involve a certain amount of backtracking. We ran both copying-based DSHOP planning system and recomputation-based DSHOP planning system on those domains. For DSHOP-*n*, we choose the *n* as 4. We compared the results with JSHOP's performance on those domains to see whether or not DSHOP could have speedup when backtracking was involved.

We randomly generated a set of Artificial domains with average branching factors (abf) = 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0. In a search tree, average branching factor is the average number of children for each node. In our experiments, there is only one solution for each problem respectively. Generally, the solution can be anywhere in a search tree. Obviously, if the solution is at the leftmost of the search tree, DSHOP cannot make any improvement with extra processors because the first worker will find the solution without any backtracking. In our experiments, we force the solutions always at the rightmost of the search trees. We want to evaluate how the performance of DSHOP changes with different amounts of backtracking involved during the planning process.

We also want to test how DSHOP performs with larger number of processors when the average branching factor is high.

4.3 Results: DSHOPC-∞ vs. DSHOPR-∞

We collected the elapsed time, exchanged message size, recomputing time, actual working time, idle time, and data format converting time for each test run. For the DSHOP, the elapsed time is calculated from the time the manager sends out the first message to one of the workers until the first message is received with a solution from one of the workers. For the JSHOP, the elapsed time is calculated from the time the planner starts searching until a solution is found. Time spent on reading and converting the input domain and problem text files is not included. All the timings (including JSHOP, DSHOPC-∞, DSHOPR-∞, DSHOPC-*n*, DSHOPR-*n*) were performed on SHARCNET.

In our experiments, we fix the "*sendoutlevel*" as 4 for DSHOP-*n* systems.

Figure 4.3.1 shows the elapsed times of DSHOPC-∞ (with two processors: one manager, one worker), DSHOPR-∞ (with two processors: one manager, one worker), and JSHOP working on a set of Artificial domains with different average branching factors.

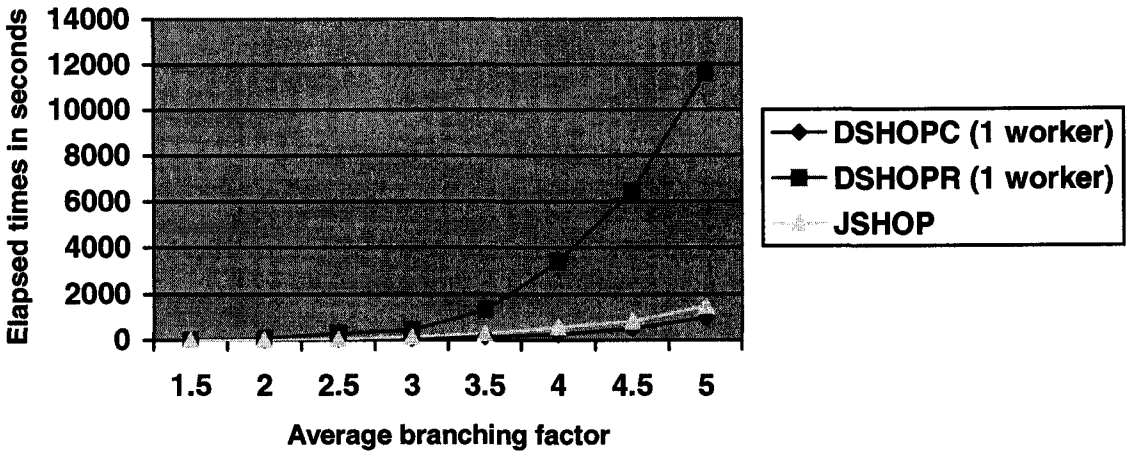

Figure 4.3.1: Elapsed times (in seconds) with one worker on artificial domains

In figure 4.3.1, the x-axis gives average branching factors of the problems, and the y-axis gives the elapsed times in seconds. As we can see from figure 4.3.1, DSHOPR-∞ did really bad in our experiments. The reason of that is because of the high percentage of recomputation (up to 90%). In our experiments, for all of the problems, DSHOPC-∞ ran faster than JSHOP even with only one worker.

We show the speedups of DSHOPC-∞ obtained with up to twelve workers for each of the above eight artificial domains with different average branching factor in table 4.3.1. And we show the speedups of DSHOPR-∞ in table 4.3.2. We compare all the parallel timings with the one-worker parallel runs to calculate the speedups. In the experiments, we observed that there was a big difference between elapsed times for same problems. That's because the network load varies from time to time caused by other jobs. Since the average values reflect these variances which have nothing to do with our implementations, we choose to use the best runs for our performance analysis [34].

Artificial domains with different average branching factor	Number of workers							Total jobs
	1	2	4	6	8	10	12	
1.5	2.112	0.999 (2.11)	0.541 (3.90)	0.408 (5.18)	0.397 (5.32)	0.382 (5.53)	0.354 (5.97)	75
2.0	10.825	5.372 (2.02)	2.740 (3.95)	1.816 (5.96)	1.634 (6.62)	1.338 (8.09)	1.228 (8.82)	589
2.5	31.176	15.146 (2.06)	7.743 (4.03)	5.153 (6.05)	4.210 (7.41)	3.386 (9.21)	3.075 (10.14)	1632
3.0	59.718	29.229 (2.04)	14.360 (4.16)	9.707 (6.15)	7.556 (7.90)	6.141 (9.72)	5.411 (11.04)	3404
3.5	139.555	69.087 (2.02)	32.155 (4.34)	23.978 (5.82)	19.518 (7.15)	17.229 (8.10)	15.136 (9.22)	6468
4.0	265.721	130.89 (2.03)	76.523 (3.47)	50.121 (5.30)	45.235 (5.87)	42.460 (6.23)	37.389 (7.11)	16152
4.5	510.359	320.455 (1.59)	201.083 (2.54)	168.942 (3.02)	156.763 (3.25)	142.744 (3.57)	138.781 (3.68)	25480
5.0	978.456	631.261 (1.55)	453.332 (2.16)	378.064 (2.58)	342.131 (2.85)	326.772 (2.99)	319.405 (3.06)	50568

Table 4.3.1. DSHOPC-∞: elapsed times (s) and speedups of the Artificial domains

From the above table, we can see that the speedups are pretty good when the average branching factor is under 4.0. In our testing domains, when the average branching factor is above 4.0, the numbers of jobs become very large. When the number of jobs is huge, it seems that there is not much space for any further improvement even with more processors. The main reason for this slowdown of speedups is because in these test domains, the numbers of jobs are very large, so that the portion of the idle time increases dramatically since most of the time is spent on sending and receiving job messages

among the manager processor and the worker processors. As a centralized scheduler, the manager processor can not deal with the large amount of messages received from all the workers as quickly as the workers do their planning jobs. The manager becomes a bottleneck, and the scheduling costs too much time. We will discuss this issue in more detail in section 4.6.

Artificial domains with different average branching factor	Number of workers							Total jobs
	1	2	4	6	8	10	12	
1.5	9.882	5.558 (1.78)	3.118 (3.17)	2.407 (4.11)	2.279 (4.34)	1.790 (5.52)	1.645 (6.01)	75
2.0	103.168	51.306 (2.01)	25.864 (3.99)	17.683 (5.83)	13.121 (7.86)	10.94 (9.43)	8.991 (11.47)	589
2.5	313.252	156.812 (2.00)	78.6 (3.99)	52.404 (5.98)	39.121 (8.01)	33.023 (9.48)	27.237 (11.50)	1632
3.0	451.284	223.707 (2.02)	112.061 (4.02)	74.823 (6.03)	56.013 (8.06)	45.342 (9.95)	37.875 (11.92)	3404
3.5	1326.09	630.891 (2.10)	309.513 (4.28)	206.284 (6.43)	152.507 (8.69)	124.453 (10.66)	103.734 (12.78)	6468
4.0	3412.18	1612.28 (2.12)	787.383 (4.33)	513.687 (6.64)	342.162 (9.97)	300.047 (11.37)	258.79 (13.19)	16152
4.5	6468.66	2623.36 (2.47)	1373.38 (4.71)	949.28 (6.81)	647.052 (9.99)	549.169 (11.79)	450.513 (14.36)	25480
5.0	11588.4	4672.45 (2.48)	2461.1 (4.71)	1516.41 (7.64)	1096.99 (10.56)	948.922 (12.21)	749.694 (15.46)	50568

Table 4.3.2. DSHOPR-∞: elapsed times (s) and speedups of the Artificial domains

From above table, we see that although the speedups are very good in DSHOPR-∞, the average elapsed time is much higher than in DSHOPC-∞. As we said previously, the reason for that is because of the high percentage of recomputation (up to 90%). We will discuss this issue in more detail in section 4.7.

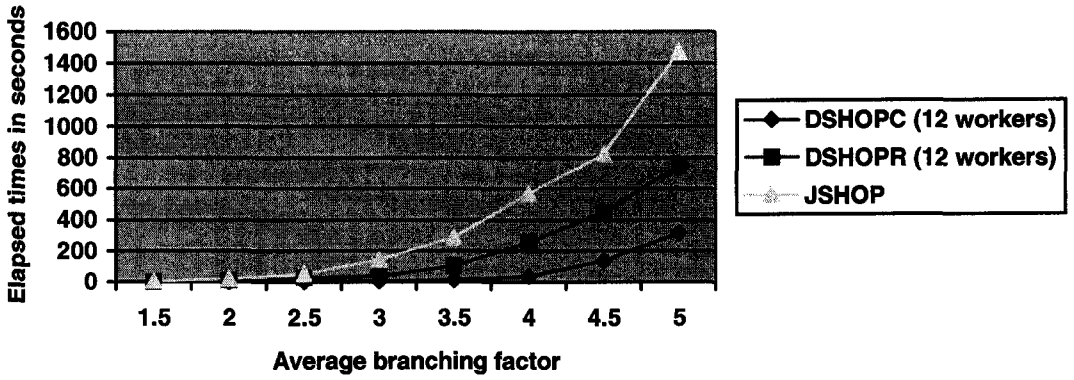

Figure 4.3.2. Elapsed times (in seconds)

Figure 4.3.2 shows the elapsed times of DSHOPC-∞ (with 12 workers), DSHOPR-∞ (with 12 workers), and JSHOP working on a set of Artificial domains. The x-axis gives the average branching factors, and the y-axis gives the elapsed times in seconds.

4.4 Results: DSHOPC-4 vs. DSHOPR-4

To reduce the high scheduling cost in DSHOPC-∞, we fixed the "*sendoutlevel*" as 4 in DSHOPC-4 to send out the alternatives only at the first 4 levels of the search tree and to let each worker explore the rest of the sub-tree by itself respectively. In this way, we reduced the number of jobs that the manager needs to deal with.

To reduce the high recomputation cost in DSHOPR-∞, we fixed the "*sendoutlevel*" as 4 in DSHOPR-4 to send out the alternatives only at the first 4 levels of the search tree

and to let each worker explore the rest of the sub-tree by itself respectively. In this way, we reduced the recomputation cost.

Figure 4.4.1 shows the elapsed times of DSHOPC-4 (with two processors: one manager, one worker), DSHOPR-4 (with two processors: one manager, one worker), and JSHOP working on a set of Artificial domains with different average branching factors.

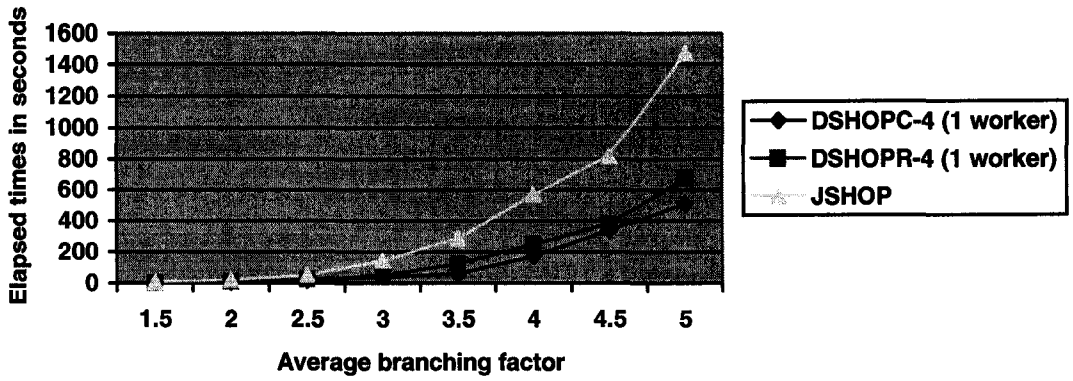

Figure 4.4.1. Elapsed times (in seconds)

In figure 4.4.1, the x-axis gives the average branching factors of the problems, and the y-axis gives the elapsed times in seconds. As we can see from figure 4.4.1, DSHOPC-4 and DSHOPR-4 did pretty good in our experiments. In our experiments, for all of the problems, DSHOPC-4 and DSHOPR-4 ran faster than JSHOP even with only one worker.

We show the speedups of DSHOPC-4 obtained with up to twelve workers for each of the above eight artificial domains with different average branching factor in table 4.4.1. And we show the speedups of DSHOPR-4 in table 4.4.2. We compare all the parallel timings with the one-worker parallel runs to calculate the speedups.

Artificial domains with different average branching factor	Number of workers							Total jobs
	1	2	4	6	8	10	12	
1.5	2.068	1.172 (1.76)	1.107 (1.87)	0.916 (2.26)	0.770 (2.69)	0.743 (2.78)	0.708 (2.92)	12
2.0	5.705	3.136 (1.82)	2.601 (2.19)	2.452 (2.33)	2.108 (2.71)	2.054 (2.78)	1.924 (2.96)	15
2.5	16.207	8.875 (1.83)	7.254 (2.23)	6.721 (2.41)	5.949 (2.72)	5.688 (2.85)	5.302 (3.05)	68
3.0	42.808	23.415 (1.83)	18.868 (2.27)	17.411 (2.46)	15.633 (2.74)	14.873 (2.88)	14.045 (3.05)	81
3.5	57.807	31.068 (1.86)	25.034 (2.31)	23.132 (2.50)	20.729 (2.79)	19.884 (2.91)	18.836 (3.07)	204
4.0	174.484	93.179 (1.87)	74.585 (2.34)	69.599 (2.51)	61.923 (2.82)	58.879 (2.96)	56.924 (3.07)	256
4.5	340.643	172.167 (1.98)	125.948 (2.70)	111.995 (3.04)	107.247 (3.17)	105.474 (3.23)	96.778 (3.52)	570
5.0	512.323	247.465 (2.07)	180.169 (2.84)	167.264 (3.06)	160.411 (3.19)	152.994 (3.35)	143.18 (3.58)	625

Table 4.4.1. DSHOPC-4: elapsed times (s) and speedups of the Artificial domains

From table 4.4.1, we observe that DSHOPC-4 did better than DSHOPC-∞ on problems with abf = 4.5, and 5.0, but the average speedup was not as good as in DSHOPC-∞ on problems with abf = 1.5, 2.0, 2.5, 3.0, 3.5, and 4.0. When we fix the "*sendoutlevel*", the number of messages sent by DSHOPC-*n* is less (only send out top levels of the tree), so the scheduling problem in DSHOPC-∞ is solved. That's why DSHOPC-4 did better than DSHOPC-∞ on problems with abf = 4.5, and 5.0. On the

other hand, when we fix the "*sendoutlevel*", we are less able to distribute work when "*sendoutlevel*" is small leading to more idle processors.

Artificial domains with different average branching factor	Number of workers							Total jobs
	1	2	4	6	8	10	12	
1.5	2.078	1.082 (1.92)	0.738 (2.81)	0.684 (3.04)	0.596 (3.48)	0.524 (3.96)	0.498 (4.17)	12
2.0	10.545	5.325 (1.98)	3.720 (2.83)	3.438 (3.08)	2.981 (3.54)	2.629 (4.01)	2.332 (4.52)	15
2.5	24.923	12.589 (1.98)	8.775 (2.84)	8.013 (3.11)	6.615 (3.76)	6.049 (4.12)	5.577 (4.46)	68
3.0	48.643	24.201 (2.01)	16.832 (2.89)	14.807 (3.28)	12.835 (3.79)	11.807 (4.12)	10.785 (4.51)	81
3.5	124.883	61.824 (2.02)	42.332 (2.95)	36.225 (3.45)	32.608 (3.83)	30.155 (4.14)	27.386 (4.56)	204
4.0	251.801	122.899 (2.04)	72.356 (3.48)	61.128 (4.11)	55.248 (4.55)	52.127 (4.83)	49.936 (5.04)	256
4.5	379.831	160.397 (2.37)	106.574 (3.56)	89.163 (4.26)	80.684 (4.71)	78.629 (4.83)	74.565 (5.10)	570
5.0	676.265	268.359 (2.52)	170.773 (3.96)	140.877 (4.80)	128.273 (5.27)	121.327 (5.57)	114.516 (5.91)	625

Table 4.4.2. DSHOPR-4: elapsed times (s) and speedups of the Artificial domains

From table 4.4.2, we observe that DSHOPR-4 did much better than DSHOPR-∞ on all the problems since it reduced a lot of recomputation cost. We will compare the recomputation cost between DSHOPR-∞ and DSHOPR-4 in section 4.7.

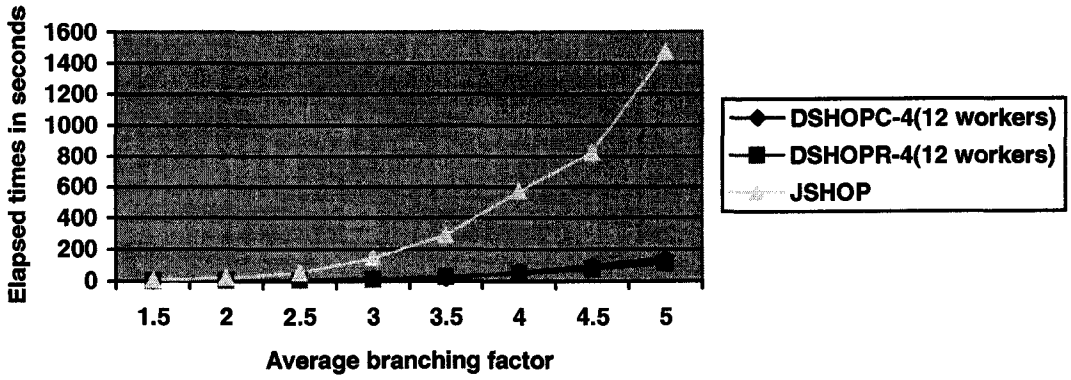

Figure 4.4.2. Elapsed times (in seconds)

Figure 4.4.2 shows the elapsed times of DSHOPC-4 (with 12 workers), DSHOPR-4 (with 12 workers), and JSHOP working on a set of Artificial domains. The x-axis gives the average branching factors, and the y-axis gives the elapsed times in seconds.

In our experiments, the performances of DSHOPC-4 and DSHOPR-4 are pretty close. Although with one worker, DSHOPC-4's performance is a little bit better than DSHOPR-4's, the average speedup of DSHOPR-4 is a little bit better than DSHOPC-4. In next section, we will compare the performances of these two systems when the value of "*sendoutlevel*" changes.

4.5 Results: DSHOPC-*n* vs. DSHOPR-*n*

We try to reduce the high scheduling cost in DSHOPC-∞ and the high recomputation cost in DSHOPR-∞ by fixing the "*sendoutlevel*" in DSHOPC-*n* and DSHOPR-*n*. In the previous section, we chose *n* as 4. We also want to know how DSHOPC-*n* and DSHOPR-*n* perform when the value of *n* differs.

Figure 4.5.1, 4.5.2, 4.5.3, 4.5.4, 4.5.5, 4.5.6, and 4.5.7 show DSHOPC-*n* and DSHOPR-*n*'s performance on one of the problems (abf = 4.5) with N = 1, 2, 4, ..., 12 workers when n = 1, 2, 3, 5, 6, 7, 8 ("*sendoutlevel*") respectively.

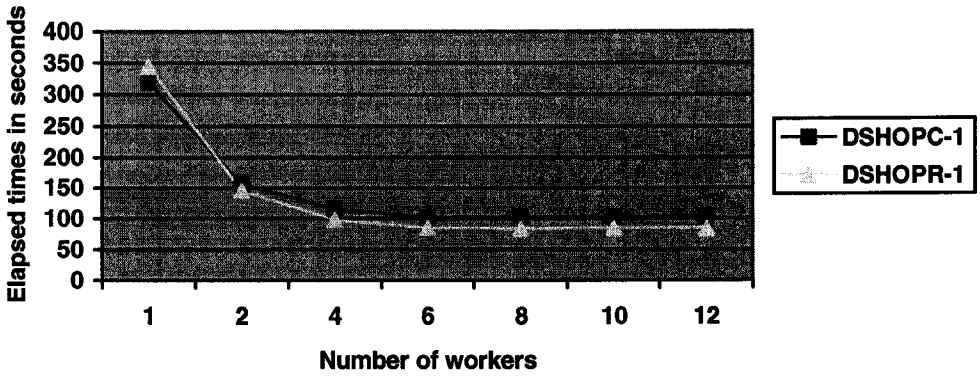

Figure 4.5.1: DSHOPC-1 vs. DSHOPR-1 on the problem with abf = 4.5

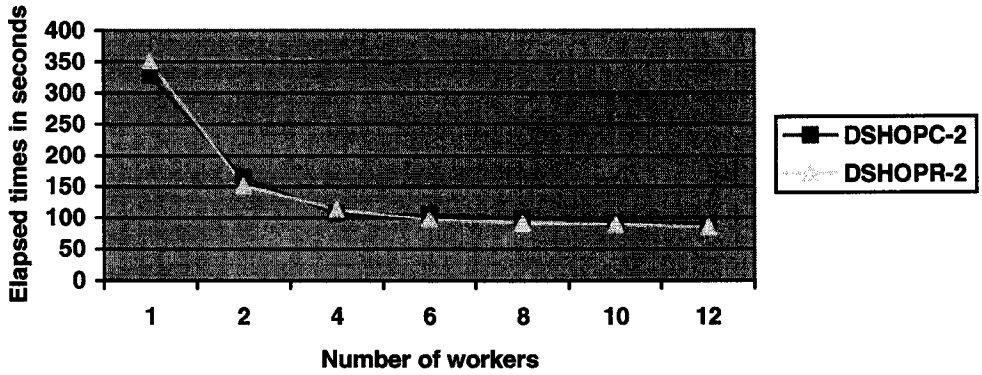

Figure 4.5.2: DSHOPC-2 vs. DSHOPR-2 on the problem with abf = 4.5

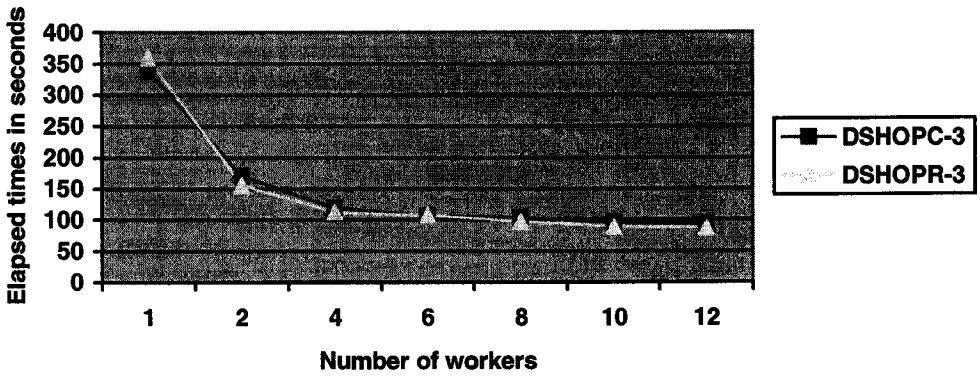

Figure 4.5.3: DSHOPC-3 vs. DSHOPR-3 on the problem with abf = 4.5

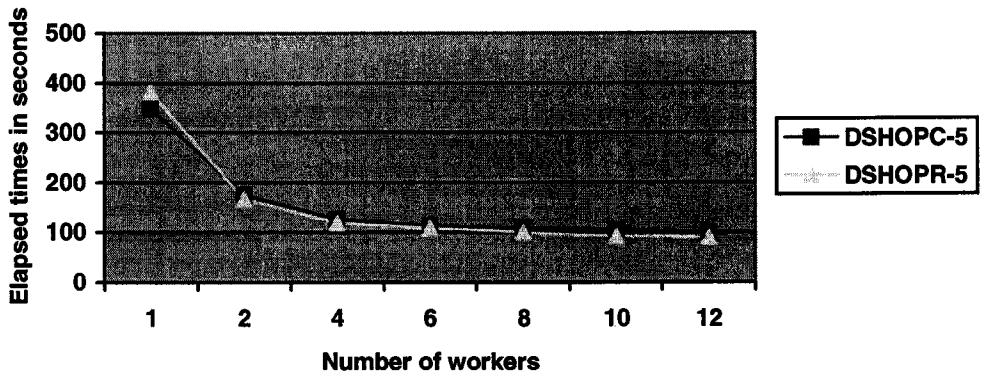

Figure 4.5.4: DSHOPC-5 vs. DSHOPR-5 on the problem with abf = 4.5

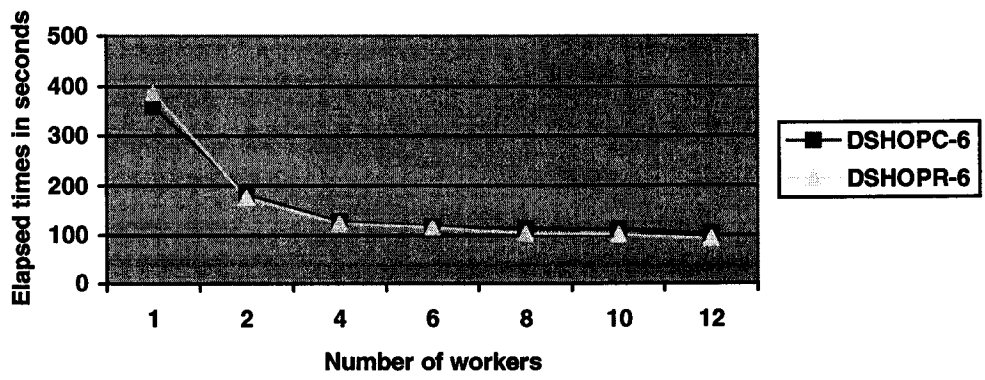

Figure 4.5.5: DSHOPC-6 vs. DSHOPR-6 on the problem with abf = 4.5

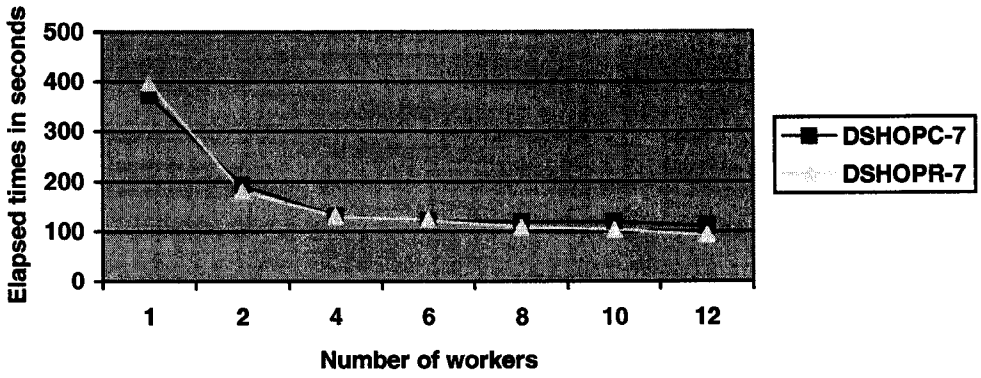

Figure 4.5.6: DSHOPC-7 vs. DSHOPR-7 on the problem with abf = 4.5

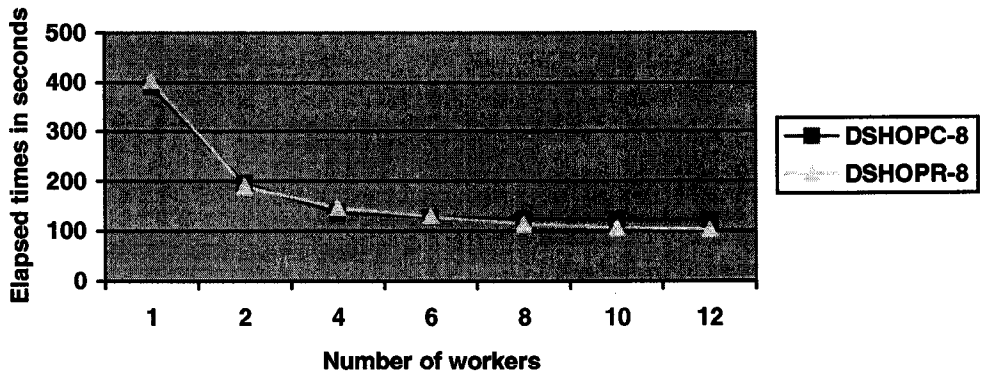

Figure 4.5.7: DSHOPC-8 vs. DSHOPR-8 on the problem with abf = 4.5

From the above five figures, we can see that in our experiments, DSHOPC-*n* and DSHOPR-*n* always have similar performances when the value of *n* differs. When *n* increases, the communication and scheduling overheads increase in DSHOPC-*n*, and the recomputation overheads increase in DSHOPR-*n*. On the other hand, when *n* decreases, there is less work that can be distributed, so the idle time may increase. Depending on the average branching factor, there may be a tradeoff formula.

4.6 Communication overhead

For DSHOPC-*n*, the main execution overheads are due to scheduling, communication and data format converting.

As we observed in section 4.4, for DSHOPC-∞, when the number of jobs is high (e.g., 25480, 50568), it seems that there is not much room for any further improvement even with more processors. In the following tables, we analysis the DSHOPC-∞ performance in the artificial domain with average branching factor 4.5 and 5.0 respectively. We give the average working time, and the data format converting time. The column of "Idle" is calculated by the average time of each worker waiting for receiving job messages from the manager. The "Idle" time actually consisted of two parts: scheduling, and communication. We also give the average percentage of elapsed time spent in different activities by all the workers.

We did experiments to test the message exchange speed on SHARCNET. In the experiments, one processor kept sending a message of size 10,000 bytes to another processor 10,000 times. We ran this program 50 times, and the average elapsed time was 1.289 seconds. So it meant that it cost 1.289 / 10,000 seconds (i.e., 0.1289 milliseconds) to transfer a 10,000 bytes message from one processor to another. In our experiments, the average message size in DSHOPC-∞ was under 500 bytes. So we think the communication overhead in DSHOPC-∞ is quite low. The high percentage of "Idle" time is mainly caused by the scheduling.

One thing to be noted here is that the above communication cost was measured based on two processors on two different nodes. On SHARCNET, one node is a four-processor, 833Mhz, Alpha SMP (symmetric multi-processors) systems. If the allocated two

processors are both on a same node, the communication speed is faster than on different nodes. In our experiments of DSHOP, when the required number of processors was under or equal to 4, we would explicitly request processors on different nodes. When the required number of processors exceeded 4, it was not necessary to do so since over 4 processors could not be on one node. In fact, in our experiments, we observed that in most cases, the allocated processors were seldom on the same nodes.

Table 4.6.1 and 4.6.2 give the percentage of time spent in working, communication and scheduling, data converting for two of the problems with different numbers of workers for DSHOPC-∞.

Activity	Workers						
	1	2	4	6	8	10	12
Working	334.788 (65%)	178.709 (56%)	82.145 (41%)	54.383 (32%)	42.167 (27%)	28.928 (20%)	24.802 (17%)
Idle	89.639 (18%)	93.445 (29%)	97.621 (48%)	100.331 (60%)	103.573 (66%)	105.742 (74%)	107.256 (78%)
Data converting	85.932 (17%)	48.301 (15%)	21.317 (11%)	14.228 (8%)	11.023 (7%)	8.074 (6%)	6.723 (5%)

Table 4.6.1: DSHOPC-∞: % Time spent in activities for the problem with abf = 4.5

Activity	Workers						
	1	2	4	6	8	10	12
Working	630.529 (65%)	350.821 (56%)	193.798 (43%)	127.338 (34%)	94.871 (28%)	77.090 (23%)	63.136 (20%)
Idle	198.126 (20%)	204.271 (32%)	211.821 (47%)	219.436 (58%)	224.129 (65%)	231.483 (71%)	240.135 (75%)
Data converting	149.801 (15%)	76.169 (12%)	47.713 (10%)	31.290 (8%)	23.131 (7%)	18.199 (6%)	16.134 (5%)

Table 4.6.2: DSHOPC-∞: % Time spent in activities for the problem with abf = 5.0

As we can see from the above table, the actual working time and data format converting time decrease when the number of workers increases, but the idle time increases. The reason that the idle time remains so high is because the number of job messages is very large. For each worker, most of the time is spent on receiving job messages from the manager, and the manager processor has to deal with (receive and send out) the large amount of messages, so that the manager becomes the bottleneck. No matter how many workers have been allocated, the time that the manager spends on scheduling and communication cannot be reduced. And adding more workers even slows down the scheduling process because the manager processor has to deal with more workers.

We roughly measure the communication overhead by calculating the total number of messages sent out by the manager and their sizes. In SHARCNET, the size of an integer is 4 bytes, while the size of a char is 1 byte. As can be seen in table 4.6.4, the message sizes and the numbers of jobs in DSHOPR-4 are very small.

Table 4.6.3 shows the communication overhead for both DSHOPC-∞ and DSHOPC-∞. And table 4.6.4 shows the communication overhead for DSHOPC-4 and DSHOPR-4.

Artificial domains with different average branching factor	DSHOPC DSHOPC-∞:			DSHOPR-∞		
	Average message size (bytes)	Number of jobs	Total message size (bytes)	Average message size (bytes)	Number of jobs	Total message size (bytes)
1.5	471	75	35332	46	75	3456
2.0	481	589	283332	122	589	71984
2.5	483	1632	788256	128	1632	208892
3.0	476	3404	1620304	134	3404	456132
3.5	489	6468	3162852	136	6468	879644
4.0	465	16152	7510680	137	16152	2212828
4.5	457	25480	11644360	142	25480	3618168
5.0	484	50568	24474912	148	50568	7484060

Table 4.6.3: Communication overheads in DSHOPC-∞ and DSHOPR-∞

Artificial domains with different average branching factor	DSHOPC-4			DSHOPR-4		
	Average message size (bytes)	Number of jobs	Total message size (bytes)	Average message size (bytes)	Number of jobs	Total message size (bytes)
1.5	238	12	2855	12	12	37
2.0	254	15	3805	12	15	49
2.5	255	68	17381	14	68	238
3.0	260	81	21044	14	81	284
3.5	258	204	52554	14	204	735
4.0	265	256	67920	15	256	939
4.5	258	570	147150	15	570	2136
5.0	271	625	169400	15	625	2344

Table 4.6.4: Communication overheads in DSHOPC-4 and DSHOPR-4

4.7 Recomputation overhead

For DSHOPR-∞, the main execution overheads are due to recomputation. So we want to know the percentage of time spent on recomputation. Table 4.7.1 gives the percentage of time spent in working and recomputing for one of the problems with different numbers of workers for DSHOPR-∞.

Activity	Workers						
	1	2	4	6	8	10	12
Working	8	8	8	8	8	8	8
Idle	0	0	0	0	0	0	0
Recomputation	92	92	92	92	92	92	92

Table 4.7.1: DSHOPR-∞: % Time spent for the problem with abf = 3.5

To reduce the high recomputation cost in DSHOPR-∞, we fixed the "*sendoutlevel*" as 4 in DSHOPR-4. Table 4.7.2 gives the percentage of time spent in working and recomputing for one of the problems with different numbers of workers DSHOPR-4.

Activity	Workers						
	1	2	4	6	8	10	12
Working	99	99	99	99	99	99	99
Idle	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Recomputation	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 4.7.2: DSHOPR-4: % Time spent for the problem with abf = 3.5

As we can see from table 4.7.2, the percentage of recomputation is very low. That's because we reduce the recomputation overhead by fixing the "*sendoutlevel*". But we also

observed that since we only send out the top levels of the alternatives, the workers have to explore the rest of the sub-trees entirely, which makes the working time high. If we increase the "*sendoutlevel*", the working time may be reduced, but the recomputation cost will increase. In our experiments, we choose the "*sendoutlevel*" as 4. Although it may not be the best setting for every problems with different numbers of workers, we found it did pretty good in most cases.

4.8 Discussion

The performance evaluation shows that, on average, the proposed algorithm DSHOP gives good improvement when backtracking involved in the planning process. Overall, in our experiments, DSHOPC-∞ has better performance than DSHOPR-∞, and DSHOPC-4 has similar performance with DSHOPR-4.

DSHOPC-∞ can gain significant speedup with multiple processors when the number of total jobs is not very large. The reason that DSHOPC-∞ could not get good performance in the problems with high amount of jobs is because of the scheduling bottleneck. The revised version DSHOPC-4 solved this problem by fixing the "*sendoutlevel*". But on the other hand, when we fix the "*sendoutlevel*", we are less able to distribute work when "*sendoutlevel*" is small leading to more idle processors.

The original DSHOPR-∞ did not get good performance because of the high percentage of recomputation cost. The revised version DSHOPR-4 reduced a lot of recomputation overhead and solved the scheduling problem by fixing the "*sendoutlevel*". This strategy is especially useful when the number of total jobs is large. DSHOPR-4 can gain good speedup with multiple processors when the number of total jobs is large.

CHAPTER 5. RELATED WORK

Distributed planning refers to an environment in which planning activity is distributed across multiple agents, processes, or sites [11]. As we discussed previously, distributed planning can be categorized into centralized planning for distributed plans, distributed planning for centralized plans, and distributed planning for distributed plans. DSHOP is in the category of distributed planning for centralized plans.

In DSHOP system, the planning process is distributed among multiple processors, each of which works on finding a complete plan. There are several different approaches to distributed planning for centralized plans. In this chapter, we will discuss some related work, and compare our approach to the others.

5.1 A-SHOP

Dix and his colleagues integrated the SHOP planning system with the IMPACT [16] multi-agent environment into A-SHOP (an agentized version of SHOP) algorithm [14].

IMPACT is a platform for agents collaborating together. In IMPACT, agents communicate with other agents by sending and receiving messages through the network. In IMPACT, an agent is a program supporting behaviors such as ongoing execution, intelligence, mobility, reactivity, communication planning, and more. Each agent has a set of associated actions such as send messages, create file, modify request and execute, etc. An IMPACT agent consists of:

- a set of data types
- a set of functions manipulating those types

- a set of actions
- a set of action constraints
- an agent program

The agentization procedure is a methodology for transforming a program to an agent. It should do the following:

- describe data types manipulated by program
- describe I/O types of function calls
- select/define actions that can be executed by agent
- define action constraints
- define agent's agent program

For example, SHOP can be agentized to a planning agent in IMPACT. Figure 5.1.1 illustrates an IMPACT architecture.

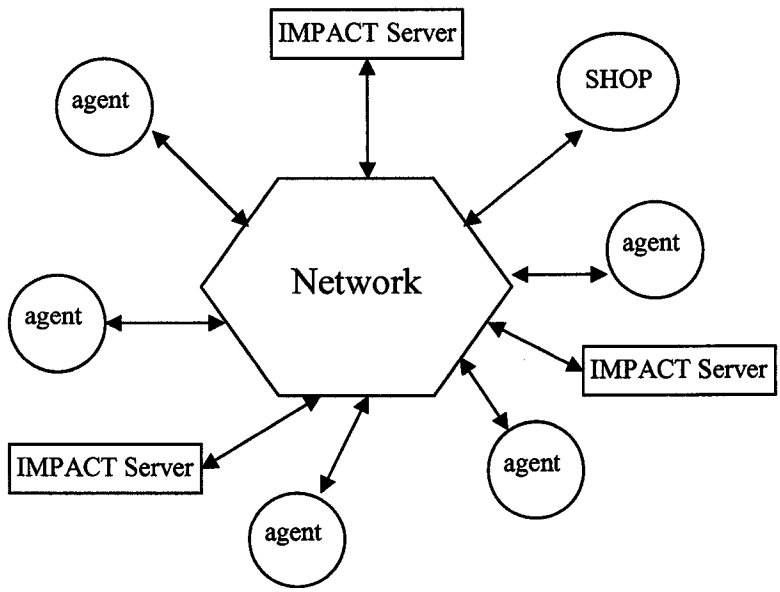

Figure 5.1.1: SHOP as a planning agent in IMPACT

In IMPACT, agents communicate with other agents through the network. Not only can they send out and receive messages from other agents, they can also ask the IMPACT server to find out services that other agents offer [14].

A-SHOP is the IMPACT version of SHOP. It plans with external information sources. Traditionally, AI planners evaluate preconditions internally. In A-SHOP, the preconditions of operators and methods are evaluated externally by other IMPACT agents, and after the application of operators, the states of the IMPACT agents are changed.

A-SHOP had been tested on a simplified version of the Noncombatant evacuation operations (NEO) planning domain, where data needed for the planning process is distributed and heterogeneous. These data were stored in other agents. Once the A-SHOP planning agent needed these data, it sent a request to other agents. The experiments showed that most of the time was spent on communication with those agents that carried the data.

A-SHOP just allows A-SHOP to gather information from distributed sources and communicate with other agents. They have not yet implemented multiple copies of A-SHOP running concurrently.

5.2 DSIPE

Corkill's distributed version of Sacerdoti's NOAH [46] planner was one of the earliest efforts in distributed HTN planning [9]. In the distributed NOAH, each distributed process has its own sub-goal to accomplish, and maintains a partial view of each other's sub-plans. Just like SIPE is conceptually descended from NOAH, Desjardins and

Wolverton's distributed SIPE (system for interactive planning and execution) [13] is conceptually descended from Corkill's distributed NOAH. DSIPE extends the ideas in distributed NOAH by focusing on efficient communication among multiple planning processors and on creating a common partial view of other planning processors' sub-plans for each planning processor. DSIPE is a fully-implemented distributed HTN planning system, and has been demonstrated to end users in the US Maritime planning community [13].

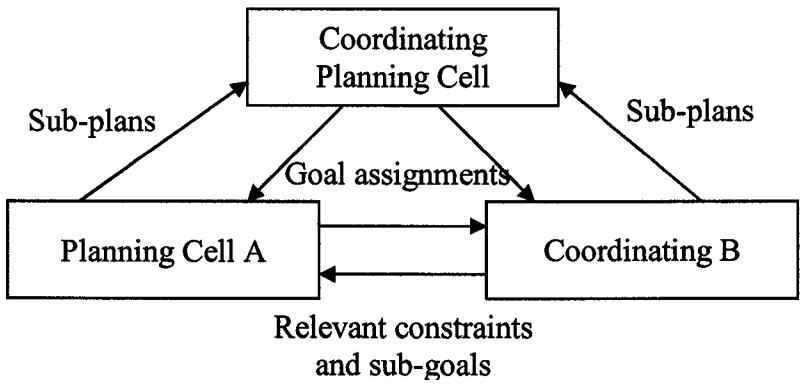

Figure 5.2.1: DSIPE architecture

DSHOP and DSIPE have something in common. For example, DSHOP is a distributed version of SHOP. DSIPE is a distributed version of SIPE-2. SHOP and SIPE are both HTN planners. And like DSHOP, multiple copies of the DSIPE planner run on separate processors that are connected across a network. But there are still several differences between DSHOP and DSIPE:

- DSHOP is performing the OR-parallel computation, while DSIPE is performing the AND-parallel computation.

- In DSIPE, distributed processes communicate with each other via message passing using KQML (knowledge query message language), while in DSHOP using MPI.
- In DSIPE, the manager partitions the sub-goals among workers, and the workers expand their sub-plans separately. At the end of the planning process, the manager merges the sub-plans together to a complete plan. While in DSHOP, each worker works on finding a complete plan, no sub-goals. Our solution is simpler, and reduces the communication overhead.
- In DSIPE, workers can communicate with each other. Each worker has a view of how other workers' sub-plans related to its local planning decisions. While in DSHOP, workers only communicate with the manager, and they do not need to communicate with each other.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

Planning has been an important subject in the area of AI for over three decades. Planning is the problem of seeking a series of actions that will accomplish a desired goal (that is, a plan).

Traditional centralized AI planning may not meet the need for solving problems in dynamic, complex, real-world domains. It is natural to think of using multiple processes working together to speed up the planning procedure. Distributed AI Planning has been developed over more than twenty years, and much research is concerned with Distributed HTN planning due to its hierarchical structure.

The distributed HTN planning architecture should provide a coordination method, a communication structure, and distributed versions of planning algorithms.

Although distributed HTN planning has been the focus of research for years, it is a still-maturing field. Distributed HTN planning involves many issues such as synchronization of multi agents, task decomposition, task allocation, conflict resolution and negotiation, information management, and plan merging.

6.1 Conclusions

We have developed a distributed version of SHOP (DSHOP). We have implemented a copying-based DSHOPC-∞ planning system, a recomputation-based DSHOPR-∞ planning system, DSHOPC-n planning system, and DSHOPR-n planning system (n can vary, in our experiments, we chose n as 4). All DSHOP systems ran on SHARCNET, and used the message-passing model to allow multiple processes to communicate. DSHOP

adopts manager/worker architecture, in which one process functions as a manager that distributes the jobs to the other processes, which are workers. The manager and workers communicate with each other via message-passing using send and receive functions in MPI library. In DSHOPC-*n*, a job message consists of the information about current state, remaining task list, and partial plan, while in DSHOPR-*n*, a job message is an oracle, which is a set of integer.

From our experiments, we can see that the proposed algorithm DSHOP gives good improvement when backtracking involved in the planning process. In our experiments, DSHOPC-∞ could not get good performance in problems with a high number of jobs because of the scheduling bottleneck, so we tried to solve this problem by fixing the "*sendoutlevel*" in DSHOPC-4. In our experiments, DSHOPR-∞ did not do well due to the high percentage of recomputation overhead (up to 90%), so we used DSHOPR-4 to reduce the recomputation overhead. DSHOPC-∞ has better performance than DSHOPR-∞, and DSHOPC-4 has similar performance with DSHOPR-4.

From our experiments, we also learn that in DSHOPC-4 and DSHOPR-4, for a given problem, the ratio of speedup would slow down once the allocated processors exceeds a certain number. The reason is that since we only send out the top levels of the alternatives, the workers have to explore the rest of the sub-trees entirely, which makes the working time remain high.

The results of our experiments seem satisfying for our current un-optimized state of DSHOP system. Moreover, DSHOP system is designed to rely on processor-processor message passing and local memories. This makes our system scalable, so that it can be easily extended to deal with large problems with a large number of processors.

Although we have developed our formalism only for SHOP, we believe that a similar approach could be used to implement the distributed version of other AI planners.

6.2 Future work

Although DSHOP can not gain much speedup on both BlocksWorld and Logistics problems due to the fact that there is no backtracking involved during the planning process, DSHOP still has the potential for speedup. We ran the experiments in the randomly generated artificial domains that had backtracking involved. The experiments show that DSHOP can have significant speedup against JSHOP with backtracking involved.

As we discussed previously, in the BlocksWorld and Logistics problems, the reason that DSHOP could not have much improvement is because there was no backtracking. And the reason that the SHOP planning process did not involve any backtracking is due to the very good search-control knowledge defined in the domain descriptions. So instinctively, we think that if we re-specify the domains for BlocksWorld and Logistics problems and provide weaker or looser search-control knowledge to introduce more alternative choice-points, DSHOP can make use of multiple processors to speedup the planning process. In other words, there is a knowledge/processor tradeoff. It means that DSHOP can make the domain specification easier.

We implemented the basic DSHOPC and DSHOPR systems. We also implemented a revised version of DSHOPC, which is DSHOPC-*n* system, and a revised version of DSHOPR, which is DSHOPR-*n* system (*n* can vary, in our experiments, we chose *n* as 4). There are a number of optimizations that can be made to improve the performance. For

instance, in our implementations, we used one centralized manager for scheduling. When the numbers of workers and jobs increase, a single manager would become a bottleneck. To solve this problem, we can divide the allocated processors into several groups. Each group has its own sub-manager, and each group is only responsible for a subset of the parallel choice-points [34].

We also can use incremental recomputation in DSHOPR as another way to reduce the amount of recomputation for each worker processor. For example, in [34], a partial incremental recomputation has been implemented, in which a worker processor does not need to recompute from the initial point to repeat those deterministic steps when it fails a job, it only needs to recompute from the first choice-point.

REFERENCES

[1] J.L. Ambite, C.A. Knoblock, Planning by rewriting, in: *Journal of Artificial Intelligence Research 15,* (2001) pp. 207-261.

[2] F. Bacchus, F. Kabanza, Using temporal logics to express search control knowledge for planning, in: *2000 Elsevier Science,* (2000) pp.123-191.

[3] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, P. Traverso, MBP: a Model Based Planner, in: *Proc. IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information,* (2001).

[4] P. Bertoli, M. Pistore, M. Roveri, Planning as Model Checking - AIPS'02 Hands-On Tutorial, at: *http://sra.itc.it/tools/mbp/AIPS02-tutorial.html,* (2002).

[5] A. Blum, M. Furst, Fast Planning Through Planning Graph Analysis, in: *Artificial Intelligence vol 58,* (1997) pp. 281-300.

[6] B. Bonet, H. Geffner, Planning as heuristic search: New results. In *Proceedings of the 5th European Conference on Planning (ECP'99),* (1999) pp. 360-372.

[7] C. Boutilier, T. Dean, S. Hanks, Decision-theoretic planning: Structural assumptions and computational leverage, in: *Journal of Artificial Intelligence Research,* (1999) pp. 1-94.

[8] W.F. Clocksin, The DelPhi Multiprocessor Inference Machine, in: *Proc. Of the 4th U.K. Conf. On Logic Prog.,* (1992) pp. 189-198.

[9] D.D. Corkill, Hierarchical planning in a distributed environment, in: *proceedings of the 6th International Joint Conference on AI. San Francisco, Morgan Kaufmann,* (1979).

[10] K. Currie, A. Tate, O-Plan: The Open Planning Architecture, in: *Artificial Intelligence Vol. 52*, (1991) pp. 49-86.

[11] M.E. Desjardins, E.H. Durfee, C.L. Ortiz, Wolverton, M.J. A Survey of Research in Distributed, Continual Planning, in: *AI Magazine 4,* (2000) pp. 13-22.

[12] E.H. Durfee, Distributed problem solving and planning, *in Multiagent Systems and Applications, MIT Press, Cambridge,* (1999) pp.118-149.

[13] M.E. Desjardins, M.J. Wolverton, Coordinating planning Activity and Information Flow in a Distributed Planning System, in: *AI Magazine*, (2000).

[14] J. Dix, H. Munoz-Avila, D. Nau, L. Zhang, IMPACTing SHOP: Putting an AI Planner into a Multi-Agent Environment, in: *Annals of Mathematics and AI*, (2000).

[15] S. Edelkamp, M. Helmert, The Model Checking Integrated Planning System (MIPS), in: *American Association for Artificial Intelligence*, (2000).

[16] T. Eiter, V. S. Subrahmanian, G. Pick, Heterogeneous Active Agents I: Semantics, *Artificial Intelligence vol 59,* (1999) pp. 179-255.

[17] K. Erol, J. Hendler, D. Nau, UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning, in: *Artificial Intelligence Planning System: proceeding of the second international conference,* (1994) pp. 249-254.

[18] K. Erol, J. Hendler, D. Nau, Semantics for Hierarchical Task-Network Planning, in: *Technical report CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9*, (1994).

[19] K. Erol, J. Hendler, D. Nau, HTN Planning: Complexity and Expressivity, in: *AAAI-94*, (1994).

[20] R.E. Fikes, N.J. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, in: *Artificial Intelligence vol 32,* (1971) pp. 189-208.

[21] T. Finin, Y. Labrou, J. Mayfield, KQML as an agent communication language, *in Software Agents. Cambridge, MA: MIT Press*, (1997).

[22] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, in: *Massachusetts Institute of Technology*, (1999).

[23] P. Haddawy, M. Suwandi, Decision-theoretic refinement planning using inheritance abstraction, in: *Hammond K, ed. Proceedings of the Second International Conference on Artificial Intelligence Planning Systems,* (1994) pp. 266-271.

[24] K.J. Hammond, CHEF: A model of case-based planning, in: *AAAI Proceedings*, (1986) pp. 261-271.

[25] B. Janssen, M. Spreitzer, D. Larner, C. Jacobi, ILU 2.0alpha 2 reference manual, *in Technical report, Xerox PARC, Palo Alto, CA,* (1998).

[26] H. Kautz, B. Selman, Blackbox: A SAT-technology planning system, at *http://www.cs.washington.edu/homes/kautz/blackbox/*, (1998).

[27] H. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: *Proceedings of the 17th National Conference of the American Association for Artificial Intelligence(IJCAI-99),* (1999) pp. 318-325.

[28] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning graphs to an ADL subset, in: *ECP-97*, (1997) pp. 273-285.

[29] R. Kurzweil, The age of Intelligent Machines, *MIT Press, Cambridge, Massachusetts,* (1990).

[30] N. Kushmerick, S. Hanks, D. Weld, An Algorithm for Probabilistic Planning, in: *Artificial Intelligence*, (1995) pp. 239-286.

[31] D. Long, M. Fox, Efficient implementation of the plan graph in STAN, in: *J. Artificial Intelligence Res. 10* (1999) pp. 87-115.

[32] A.D. Mali, S. Kambhampati, Distributed planning, in: *Encyclopedia of Distributed Computing,* (2002).

[33] D. McDermott and the AIPS-98 Planning Competition Committee, PDDL - The Planning Domain Definition Language, at: *ftp://ftp.cs.yale.edu/pub/mcdermott/software /pddl.tar.gz*, (1998).

[34] S. Mudambi, J. Schimpf, Parallel CLP on Heterogeneous Networks, in: *Proc. of the 11th ICLP, MIT Press*, (1994) pp. 124-141.

[35] H. Munoz-Avila, D. Aha, L. Breslow, D. Nau, HICAP: an interactive case-based planning architecture and its application to noncombatant evacuation operations, in: *IAAI-99, (*1999) pp. 870-875.

[36] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila, SHOP: Simple Hierarchical Ordered Planner, in: *IJCAI-99*, (1999) pp. 968-973.

[37] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila, SHOP and M-SHOP: Planning with Ordered Task Decomposition, in: *Tech. Report CS TR 4157, University of Maryland, College Park, MD*, (2000).

[38] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila, The SHOP Planning System, in: *AAAI-01*, (2001).

[39] N. Nilsson, Principles of Artificial Intelligence. *Morgan Kauffmann,* (1980).

[40] P.S. Pacheco, Parallel Programming with MPI, *Morgan Kaufmann Publishers, Inc*, (1996).

[41] E. Pednault, ADL: Exploring the middle ground between STRIPS and the situation calculus, in: *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, (1989) pp. 324-332.

[42] J.S. Penberthy, D.S. Weld, UCPOP: A sound, complete, partial order planner for ADL, in: *Proceeding of Third International Conference of Principles of Knowledge Representation and Reasoning*, (1992).

[43] M. Peot, D.E. Smith, Conditional Nonlinear Planning, in: *Proceedings of the First International Conference on AI Planning Systems (AIPS-9). College Park, MD*, (1992) pp. 189-197.

[44] E. Rich, K. Knight, Artificial Intelligence. *McGraw-Hill, Inc,* (1991).

[45] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, *Prentice Hall Series in AI* , (1995).

[46] E. D. Sacerdoti, A structure for plans and behavior, *American Elsevier*, (1977).

[47] J. Schumann, R. Letz, PARTHEO: A high-performance parallel theorem prover, in: *Proc. Of CADE'90,* (1990) pp. 40-56.

[48] E. Shapiro, Or-Parallel Prolog in Flat Concurrent Prolog, in: *Journal of Logic Programming,* (1989) pp. 243-267.

[49] P. Stone, M. Veloso, Multiagent Systems: A Survey from a Machine Learning Perspective, in: *CMU-CS-97-193*, (1997).

[50] V.S. Sunderam, PVM: A Framework for Parallel Distributed Computing, *in Concurrency: Practice and Experience,* (1990) pp315-339.

[51] A. Tate, Project Planning Using a Hierarchic Non-Linear Planner, *Research Report No. 25, Department of Artificial Intelligence, University of Edinburgh*, (1976).

[52] M. Veloso, Learning by Analogical Reasoning in General Problem solving, in: *Tech. Report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA*, (1992).

[53] M. Veloso, J. Blythe, Linkability: Examining causal link commitments in partial-order planning, in: *Proc. AIPS-94*, (1994).

[54] D.S. Weld, Recent Advances in AI Planning. In *AI Magazine, 20(2),* (1999) pp. 93-123.

[55] D.E. Wilkins, Practical Planning: Extending the Classical AI Planning Paradigm, *Morgan Kaufmann, Los Altos, California*, (1988).

[56] http://www.mpi-forum.org/

[57] http:// www.sharcnet.ca

VITA AUCTORIS

NAME:	Shuyun Lu
PLACE OF BIRTH:	ShangHai, China
YEAR OF BIRTH:	1972
EDUCATION:	The First High School, BengBu, AnHui, China 1983-1989
	AnHui University, HeFei, AnHui, China 1989-1993 B.Sc.
	University of Windsor, Windsor, Ontario, Canada 2001-2004 M.Sc.